Quantifying Information Leakage of Randomized Protocols

Fabrizio Biondi, Axel Legay, Pasquale Malacaria, Andrzej Wasowski

Quantifying Information Leakage of Randomized Protocols

Fabrizio Biondi[a], Axel Legay[a], Pasquale Malacaria[b], Andrzej Wąsowski[c]

[a] *IRISA/INRIA Rennes, 263 Avenue du Général Leclerc, 35042 Rennes, France*
[b] *Queen Mary University of London, Mile End Road, E1 4NS, London, United Kingdom*
[c] *IT University of Copenhagen, 7 Rued Langgaards Vej, 2300 Copenhagen-S, Denmark*

Abstract

The quantification of information leakage provides a quantitative evaluation of the security of a system. We propose the usage of Markovian processes to model deterministic and probabilistic systems. By using a methodology generalizing the lattice of information approach we model refined attackers capable to observe the internal behavior of the system, and quantify the information leakage of such systems. We also use our method to obtain an algorithm for the computation of channel capacity from our Markovian models. Finally, we show how to use the method to analyze timed and non-timed attacks on the Onion Routing protocol.

1. Introduction

Quantification of information leakage is a recent technique in security analysis that evaluates the amount of information about a secret (for instance about a password) that can be inferred by observing a system. It has sound theoretical bases in Information Theory [1, 2]. It has also been successfully applied to practical problems like proving that patches to the Linux kernel effectively correct the security errors they address [3]. It has been used for analysis of anonymity protocols [4, 5] and analysis of timing channels [6, 7]. Intuitively, *leakage of confidential information of a program is defined as the difference between the attacker's uncertainty about the secret before and after available observations about the program* [1].

The underlying algebraic structure used in leakage quantification for *deterministic* programs is the *lattice of information* (LoI) [1]. In the LoI approach an attacker is modeled in terms of possible observations of the system she can

☆An earlier versions of this paper appeared in the 14th International Conference on Verification, Model Checking, and Abstract Interpretation (2013).

☆☆The research presented in this paper has been partially supported by MT-LAB, a VKR Centre of Excellence for the Modelling of Information Technology.

Email addresses: fabrizio.biondi@inria.fr (Fabrizio Biondi), axel.legay@inria.fr (Axel Legay), p.malacaria@qmul.ac.uk (Pasquale Malacaria), wasowski@itu.dk (Andrzej Wąsowski)

make. LoI uses an equivalence relation to model how precisely the attacker can distinguish the observations of the system. An execution of a program is modeled as a relation between inputs and observables. In this paper we follow the LoI approach but take a process view of the system. A process view of the system is a more concise representation of behavior than an observation relation. Moreover a process view does not require that the system is deterministic, which allows us to handle randomized protocols—for the first time using a generic, systematic and implementable LoI-based methodology.

We use Markov Decision Processes to represent the probabilistic partial-information semantics of programs, using the nondeterminism of the model for the choices that depend on the unknown secret. We define the leakage directly on such model. With our method we can distinguish the inherent randomness of a randomized algorithm from the unpredictability due to the lack of knowledge about the secret. We exploit this distinction to quantify leakage only for the secret, as the information leakage about the random numbers generated is considered uninteresting (even though it is information in information theoretical sense). We thus work with both deterministic and randomized programs, unlike the previous LoI approach.

We give a precise encoding of an *attacker* by specifying her prior knowledge and observational capabilities. We need to specify which of the logical states of the system can be observed by the attacker and which ones he is able to distinguish from each other. Given a program and an attacker we can calculate the leakage of the program to the attacker.

We also show how to turn the leakage computation into leakage optimization: we compute the maximum leakage over all possible prior information of attackers *ceteris paribus*, or in other words, the leakage for the worst possible attacker without specifying the attacker explicitly. This maximum leakage is known as the *channel capacity* of the system [8]. Since we are able to model a very large class of attackers the obtained channel capacity is robust. Computing channel capacity using this method requires solving difficult optimization problems (as the objective is nonlinear), but we show how the problem can be reduced to standard reward optimization techniques for Markovian models for a class of interesting examples.

Our method can be applied to finite state systems specified using a simple imperative language with a randomization construct. It can also be used for systems modeled directly as Markov Decision Processes. We demonstrate the technique using an MDP model of the known Onion Routing protocol [9], showing that we can obtain the channel capacity for a given topology from an appropriate Markov Decision Process describing the probabilistic partial information behavior of the system. Also, our behavioral view of the system allows us to encode an attacker with time-tracking capabilities and prove that such an attacker can leak more information than the canonical attacker that only observes the traffic on the compromised nodes. Timing-based attacks to the Onion Routing protocol have been implemented before [10, 11], but to our best knowledge the leakage of timing-attacks has not been quantified before.

Our contributions include:

- A method for modeling attack scenarios consisting of process models of systems and observation models of attackers, including a simple partial-observability semantics for imperative programs, so that these models can also be obtained from code.
- A definition of leakage that generalizes the LoI approach to programs with randomized choices (strictly including the class of deterministic programs), and dually the first application of the LoI approach to process specifications of systems.
- A method for computing leakage for scenarios modeled as described above. The method is fully implementable.
- A method to parametrize the leakage analysis on the attacker's prior information about the secret, to allow the computation of channel capacity by maximizing an objective characterizing leakage as a function of prior information.
- The worst-case analysis of the Onion Routing protocol when observed by non time-aware and time-aware attackers able to observe the traffic passing through some compromised nodes.

The paper proceeds as follows. Section 2 provides the core background on probabilistic systems and the LoI approach. Section 3 gives an overview of our new leakage quantification method. The non-obvious steps are further detailed in Sections 4–6. In Sect. 8 we explain how to use the method for computing channel capacity, and we use this technique to analyze leakage in the onion routing protocol against untimed and timing attacks (Sect. 9). We discuss the related work (Sect. 11) and conclude (Sect. 12).

2. Background

We often refer to deterministic, stochastic and nondeterministic behavior. We use the adjective *deterministic* for a completely predictable behavior, *stochastic* for a behavior that follows a probability distribution over some possible choices, and *nondeterministic* for a choice where no probability distribution is given.

We define common concepts in probability theory and information theory that are used throughout the paper. We refer to basic books on the subject [12, 13] for the definitions of sample space S, probability of event $P(E)$ and so on. We use $\mathcal{X}$ to denote a *discrete stochastic process*, i.e. an indexed infinite sequence of discrete random variables $(X_0.X_1, X_2, ...)$ ranging over the same sample space S. The index of the random variables in a stochastic process can be understood as modeling a concept of discrete time, so X_k is the random variable representing the system at time unit k.

2.1. Markovian Models

A discrete stochastic process is a *Markov chain* $\mathcal{C} = (C_0, C_1, C_2, ...)$ iff $\forall k \in \mathbb{N}$. $P(C_k|C_{k-1}, ..., C_0) = P(C_k|C_{k-1})$. A Markov chain on a sample space S can also be defined as follows:

Definition 1. A tuple $\mathcal{C} = (S, s_0, P)$ is a Markov Chain (MC), if S is a finite set of states, $s_0 \in S$ is the initial state and P is an $|S| \times |S|$ probability transition matrix, so $\forall s, t \in S$. $P_{s,t} \geq 0$ and $\forall s \in S$. $\sum_{t \in S} P_{s,t} = 1$.

We write $\pi^{(k)}$ for the probability distribution vector over S at time k and $\pi^{(k)}_s$ the probability $\pi^k(s)$ of visiting the state s at time k. This means that, considering a Markov chain $\mathcal{C}$ as a time-indexed discrete stochastic process $(C_0, C_1, ...)$, we write $\pi^{(k)}$ for the probability distribution over the random variable C_k. Since we assume that the chain starts in state s_0, then $\pi^{(0)}_s$ is 1 if $s = s_0$ and 0 otherwise. Note that $\pi^{(k)} = \pi_0 P^k$, where P^k is matrix P elevated to power k, and P^0 is the identity matrix of size $|S| \times |S|$.

A state $s \in S$ is *absorbing* if $P_{s,s} = 1$. In the figures we will not draw the looping transition of the absorbing states, to reduce clutter. We say that a Markov chain is *one-step* if all states except the starting state s_0 are absorbing.

Let $\xi(s, t)$ denote the *expected residence time* in a state t in an execution starting from state s given by $\xi(s, t) = \sum_{n=0}^{\infty} P^n_{s,t}$. We will write ξ_s for $\xi(s_0, s)$.

Given k Markov chains $\mathcal{C}^1 = (S^1, s^1_0, P^1)$,...,$\mathcal{C}^k = (S^k, s^k_0, P^k)$ their synchronous *parallel composition* is a MC $\mathcal{C} = (S, s_0, P)$ where S is $S^1 \times ... \times S^k$, s_0 is $s^1_0 \times ... \times s^k_0$ and $P_{s^1 \times ... \times s^k, t^1 \times ... \times t^k} = \prod_{i=1}^{k} P_{s^i, t^i}$.

A real-valued *reward function* on the transitions of a MC $\mathcal{C} = (S, s_0, P)$ is a function $R : S \times S \to \mathbb{R}$. Given a reward function on transitions, the *expected reward* $R(s)$ for a state $s \in S$ can be computed as $R(s) = \sum_{t \in S} P_{s,t} R(s, t)$, and the *expected total reward* $R(\mathcal{C})$ of $\mathcal{C}$ as $R(\mathcal{C}) = \sum_{s \in S} R(s)\xi_s$.

Definition 2. A *Markov Decision Process* (MDP) is a tuple $\mathcal{P} = (S, s_0, P, \Lambda)$ where S is a finite set of states containing the initial state s_0, Λ_s is the finite set of available actions in a state $s \in S$ and $\Lambda = \bigcup_{s \in S} \Lambda_s$, and $P : S \times \Lambda \times S \to [0, 1]$ is a transition probability function such that $\forall s, t \in S. \forall a \in \Lambda_s$. $P(s, a, t) \geq 0$ and $\forall s \in S. \forall a \in \Lambda_s$. $\sum_{t \in S} P(s, a, t) = 1$.

We we will write $s \xrightarrow{a} [P_1 \mapsto t_1, ..., P_n \mapsto t_n]$ to denote that in state $s \in S$ the system can take an action $a \in \Lambda_s$ and transition to the states $t_1, ..., t_n$ with probabilities $P_1, ..., P_n$.

We will enrich our Markovian models with a finite set $\mathtt{V}$ of natural-valued variables, and we assume that there is a very large finite bit-size M such that a variable is at most M bit long. We define an assignment function $A : S \to [0, 2^M - 1]^{|\mathtt{V}|}$ assigning to each state the values of the variables in that state. We will use the expression $\mathtt{v}(s)$ to denote the value of the variable $\mathtt{v} \in \mathtt{V}$ in the state $s \in S$.

Given a Markov chain $\mathcal{C} = (S, s_0, P)$ let a *discrimination relation* $\mathcal{R}$ be an equivalence relation over S. We use discrimination relation to quotient one-step Markov chains:

Definition 3. Given a one-step Markov chain $\mathbb{C} = (S, s_0, P)$ and a discrimination relation $\mathcal{R}$ over S, we define the *quotient* $\mathbb{C}/\mathcal{R}$ of $\mathbb{C}$ over $\mathcal{R}$ as the one-step Markov chain $\mathbb{C}/\mathcal{R} = (\bar{S}, \bar{s}_0, \bar{P})$ where

- $\bar{S}$ is the set of equivalence classes of S induced by $\mathcal{R}$;
- $\bar{s}_0$ is the equivalence class of s_0;
- for each equivalence class $\bar{s} \in S'$,

$$\bar{P}_{\bar{s}_0,\bar{s}} = \sum_{s \in \bar{s}} P_{s_0,s}$$

and $\bar{P}_{\bar{s},\bar{s}} = 1$.

2.2. Information Theory

The *entropy* of a probability distribution is a measure of the unpredictability of the events considered in the distribution [14].

Definition 4. [13] Let X and Y be two random variables with probability mass functions $p(x)$ and $p(y)$ respectively and joint probability mass function $p(x, y)$. Then we define the following non-negative real-valued functions:

- Entropy $H(X) = -\sum_{x \in X} p(x) \log_2 p(x)$
- Joint entropy $H(X, Y) = -\sum_{x \in X} \sum_{y \in Y} p(x, y) \log_2 p(x, y)$
- Conditional entropy $H(X|Y) = -\sum_{x \in X} \sum_{y \in Y} p(x, y) \log_2 p(x|y) =$
  $= \sum_{y \in Y} p(y) H(X|Y = y) = \sum_{y \in Y} p(y) \sum_{x \in X} p(x|y) \log_2 p(x|y) =$
  $= H(X, Y) - H(Y)$ (chain rule)
- Mutual information $I(X; Y) = \sum_{x \in X} \sum_{y \in Y} p(x, y) \log_2 \left(\frac{p(x,y)}{p(x)p(y)} \right) =$
  $= H(X) + H(Y) - H(X, Y) \leq \min(H(X), H(Y))$

We will sometimes write $H(\mathbf{P}(x_1), \mathbf{P}(x_2), .., \mathbf{P}(x_n))$ for the entropy of the probability distribution over $x_1, ..., x_n$. Mutual information can be generalized to two vectors of random variables $\bar{X}, \bar{Y}$ as $I(\bar{X}; \bar{Y}) = \sum_{\bar{x} \in \bar{X}} \sum_{\bar{y} \in \bar{Y}} p(\bar{x}, \bar{y}) \log_2 \left(\frac{p(\bar{x},\bar{y})}{p(\bar{x})p(\bar{y})} \right)$.

Since every state s in a MC $\mathcal{C}$ has a discrete probability distribution over the successor states we can calculate the entropy of this distribution. We will call it *local entropy*, $L(s)$, of s: $L(s) = -\sum_{t \in S} P_{s,t} \log_2 P_{s,t}$. Note that $L(s) \leq \log_2(|S|)$.

As a MC $\mathcal{C}$ can be seen as a discrete probability distribution over all of its possible traces, we can assign a single entropy value $H(\mathcal{C})$ to it. The global entropy $H(\mathcal{C})$ of $\mathcal{C}$ can be computed by considering the local entropy $L(s)$ as the expected reward of a state s and then computing the expected total reward of the chain [15]:

$$H(\mathcal{C}) = \sum_{s \in S} L(s) \xi_s$$

If a Markov chain is one-step its entropy corresponds to the local entropy of the initial state s_0.

2.3. Information Leakage

Information leakage is a quantitative measure of the security of a system. Consider a system whose behavior depends on some secret data, like an authentication or anonymity protocol. By observing the behavior of the system an attacker can infer information about the value of the secret. In information-theoretical terms, the system itself can be considered a channel transmitting information between the secret and the attacker, and the amount of information that passes through this channel corresponds to the amount of information that the attacker learns about the secret.

Before observing the behavior of the system, the attacker has some prior information about the secret, for example its size in bits. This is encoded in its prior distribution h over the values of the secret. The lack of information of the attacker on the secret before the observation of the system is quantified as the uncertainty of this prior distribution, or $U(h)$.

Then the attacker observes the output of the system, which we will call O. From this observation he will learn information about the secret; in particular his uncertainty about the secret after the observation will be defined as $U(h|O)$. The difference between the attacker's uncertainty on the secret before and after the observation is the information that he learned from the observation, thus the information *leakage* is $U(h) - U(h|O) = I(O;h)$.

Various measures have been proposed to quantify the information leakage of a system. In this work we consider *Shannon leakage*, meaning that $U(h)$ just corresponds to the entropy $H(h)$ of the probability distribution over h, $U(h|O)$ to the posterior entropy $H(h|O)$ and $I(O;h)$ to the mutual information between O and h. Other measures include *min-entropy leakage* [16, 17] when $U(h) = \max_i(h_i)$ and *guessing entropy* [7, 18] when $U(h) = \sum_i i \cdot (h_i)$ with $p(h_i) \geq p(h_2) \geq ... \geq p(h_n)$. A generalization of the leakage measures via gain function called *g-leakage* has been proposed by Alvim et al. [19]. Similarly, Boreale and Pampaloni obtained results on adaptive adversaries under a generalized uncertainty measure [20]. The computational technique proposed in this work computes Shannon leakage, but it can be adapted to other leakage measures.

$$\begin{array}{c} \\ 0 \\ 1 \end{array} \begin{array}{c} \begin{array}{cc} 0 & 1 \end{array} \\ \begin{pmatrix} 0.3 & 0.7 \\ 0.7 & 0.3 \end{pmatrix} \end{array}$$

Figure 1: Bit XOR channel matrix

As we remarked, these definitions of leakage consider the system as a channel between the secret and the output observable by the attacker. It is common to write down this channel explicitly as a *channel matrix*, a matrix $C_{i,j}$ where the element (i, j) represents the probability that the output produced will be j if the value of the secret is i. Consider a simple program that has a 1-bit secret h, generates at random a bit r with $\mathrm{P}(r = 0) = 0.3$ and outputs the exclusive OR of h and r. The channel matrix for this program is shown in Fig. 1.

$$\begin{array}{c} \\ 0 \\ 1 \end{array} \begin{array}{c} \begin{array}{cc} 0 & 1 \end{array} \\ \begin{pmatrix} 1 & 0 \\ 0.3 & 0.7 \end{pmatrix} \end{array}$$

Figure 2: Bit AND channel matrix

If the program were to calculate the logical AND of h and r instead, the channel matrix would be the one presented in Fig. 2.

A channel matrix represents an attacker's knowledge about the system as a functions from values of the secret to probability distributions over the outputs

the attacker is able to discern. In this sense it is both accounting for the system's behavior and for the attacker's ability to observe it. It is not accounting for the attacker's prior information, and thus is independent from it, meaning that two attackers with the same discriminating power and different prior information will be represented by the same channel matrix.

To compute a value for information leakage we also need the attacker's prior distribution h over the values of the secret. Once that is fixed, Shannon leakage can be computed as $H(h) - H(h|O) = H(h) - \sum_{o \in O} P(o)H(h|O = o)$.

2.4. Information Leakage Orderings

Consider a program depending on a secret h. We want to construct an ordering of attackers in which for attackers A and B, $A \sqsubseteq B$ means that attacker A is less effective in discovering information about the secret than attacked B. Intuitively this means that if A and B begin an attack having the same prior information about h, then B will discover at least as many bits of information as A.

Definition 5. Let A and B be two attackers on programs with the same secret h and sharing the same prior information $\mu(h)$. Then we define the *leakage ordering* as

$$A \sqsubseteq B \text{ iff } \forall \mu(h).L(A) \leq L(B)$$

Since the channel matrix representation encodes the posterior probabilities of the observables as a function of the possible values of the secret, we can obtain an ordering just by comparing the channel matrices of A and B. In the following we will say "attacker A" for "the channel matrix encoding of the program when observed by attacker A"; properly we are defining orderings on channel matrices, not on attackers.

Let's compare the channel matrices in Fig. 1 and 2. The former encodes an attacker that can observe the result of a logical XOR between the secret bit and a random bit r with $P(r = 0) = 0.3$, while the latter encodes an attacker that can observe the result of a logical AND between the same secret and the same random bit. Is one of the two channels more informative on the secret than the other?

In general we would expect different leakage measures to induce different leakage orderings. In fact, this depends on whether the programs we consider are fully deterministic or include some randomization mechanism. We present current results about the ordering for deterministic and non-deterministic programs.

Ordering on deterministic programs: partition refinement. Let Σ be a finite set of observables over a deterministic program $\mathcal{P}$. Consider all possible equivalence relations over Σ; each of them represents the discriminating power of an attacker. Then we define *partition refinement ordering* between attackers as follows:

Definition 6. Let A and B be two attackers and $\sim_A$, $\sim_B$ be their observational equivalence relations over Σ. We define *partition refinement ordering* $\sqsubseteq$ as

$$A \sqsubseteq B \quad \text{iff} \quad \forall \sigma_1, \sigma_2 \in \Sigma \, (\sigma_1 \sim_B \sigma_2 \Rightarrow \sigma_1 \sim_A \sigma_2) \tag{1}$$

The ordering forms a *complete lattice* over the set of all possible equivalence relations over Σ [21]: the Lattice of Information (abbreviated as `LoI`).

If $A \sqsubseteq B$ then classes in $\sim_B$ refine (split) classes in $\sim_A$, thus $\sim_B$ is the observational equivalence relation of an attacker that can distinguish more while $\sim_A$ is the observational equivalence relation an attacker that can distinguish less observables.

Importantly, if $A \sqsubseteq B$ then A leaks less information than B under many different leakage measures, including Shannon leakage, guessing entropy and min-entropy leakage [1, 22].

By equipping the set Σ with a probability distribution we can see an equivalence relation as a random variable (technically it is the set theoretical kernel of a random variable but for information theoretical purposes can be considered a random variable [1]). Hence the `LoI` can be seen as a lattice of random variables.

The connection between `LoI` and leakage can be illustrated by this simple example: consider a password checking program checking whether the user input is equal to the secret `h`. Then an attacker observing the outcome of the password check will know whether the secret is `h` or not, hence we can model the leakage of such a program with the equivalence relation $\{\{\mathtt{h}\}, \{x | x \neq \mathtt{h}\}\}$.

More generally, observations over a deterministic program $\mathcal{P}$ form an equivalence relation over the possible states of $\mathcal{P}$. A particular equivalence class will be called an observable. Hence an observable is a set of states indistinguishable by an attacker making that observation. If we consider an attacker able to observe the outputs of a program then the random variable associated to a program $\mathcal{P}$ is given by the equivalence relation on any two states σ, σ' from the universe of program states Σ defined by

$$\sigma \simeq \sigma' \iff [\![\mathcal{P}]\!](\sigma) = [\![\mathcal{P}]\!](\sigma') \tag{2}$$

where $[\![\mathcal{P}]\!]$ represents the denotational semantics of $\mathcal{P}$ [23]. Hence the equivalence relation amounts to "having the same observable output". This equivalence relation is nothing else than the set-theoretical kernel of the denotational semantic of $\mathcal{P}$ [24].

Ordering on non-deterministic programs: composition refinement. Partition refinement is based on the fact that in deterministic programs different observables induce a partition over the possible values of the secret, i.e. it is not possible for two different observables to be produced by a single value of the secret. This property does not hold with non-deterministic programs, so partition refinement is unfitting as an ordering when non-deterministic programs are involved.

We say that there is a *channel factorization* between two channels A and B if there exists a channel C such that $A = BC$, in the sense of matrix multiplication. In their work introducing the Lattice of Information [21], Landauer and Redmond noted that two channels A and B are ordered under partition refinement if and only if there exists a channel factorization between them. This ordering has

been studied by Alvim et al. [19] and by McIver et al. [25, 26] with the name *composition refinement ordering* $\sqsubseteq_o$:

Definition 7. Let A and B be channels. Define *composition refinement ordering* $\sqsubseteq_o$ as

$$A \sqsubseteq_o B \;\Leftrightarrow\; \exists C.\ A = BC$$

where C is a channel.

In general, different leakage measures induce different composition refinement orderings. To adapt leakage measures to different problems, Alvim et al. [19] proposed the *g-leakage* measure, that generalizes min-entropy leakage via arbitrary gain functions. In particular it is possible to apply gain functions that make g-leakage equivalent to Shannon leakage, min-entropy leakage, guessing entropy, etc. For this reason, results obtained on g-leakage can be generally applied to multiple leakage measures.

In particular, write $A \leq_{\mathcal{G}} B$ if A leaks less information than B under *any* gain function. Then the following result holds [19, 26]:

Lemma 1.

$$A \leq_{\mathcal{G}} B \;\Leftrightarrow\; A \sqsubseteq_o B$$

proving the soundness and completeness of the composition refinement ordering under the g-leakage measure.

In Section 6 we introduce a leakage ordering for probabilistic processes based on the technique we present in this paper. The ordering is based on Shannon leakage, and thus induces the same ordering as composition refinement with Shannon leakage.

3. Information Leakage of Markov Chains

We give an overview of the proposed method for leakage quantification, deferring the details to later sections. The method proceeds in five steps, that are all fully automatable for finite state programs. Let a *scenario* be a pair $(\mathcal{P}, \mathcal{A})$, where $\mathcal{P}$ is the system we want to analyze and $\mathcal{A}$ is an attacker. We will call $\mathcal{P}$ the *program*, even if it can be any system suitably modeled as an MDP as explained in Sect. 4.

Step 1: Define a MDP representing $\mathcal{P}$ (Sections 4, 9). We first give a probabilistic semantics to the program in the form of an MDP, in which probabilistic choices are represented by successor state distributions and branching is represented by decision states. This is more or less standard definition of operational semantics for randomized imperative programs.

Example [24]. A program has two 2-bit long variables `l` and `h`. Variable `h` is secret, while variable `l` is public, meaning that the attacker can read `l` but not `h`:

```
l = 0; while (l != h) do l = l + 1;
```

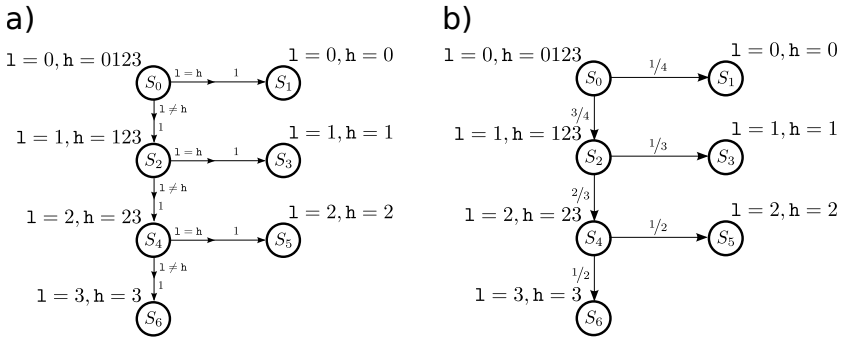

Figure 3: Simple loop example a) MDP semantics b) MC model

The MDP representing the probabilistic partial information semantics of the program is depicted in Fig. 3a. The states in which the system stops and produces an output are encoded with the absorbing states of the MDP, i.e. the states with a probability of transitioning to themselves equal to 1. In the MDP in Fig. 3a states S_1, S_3, S_5 and S_6 are absorbing states.

Step 2: Define the attacker $\mathcal{A}$. The MDP we obtained models the behavior of the process at the required abstraction level, meaning that public variables (`l`) are given values and secret variables (`h`) are given probability distributions. Now we need to also consider the model of the attacker to transform the MDP in the Markov chain modeling the process when observed by the attacker.

Definition 8. An attacker is a pair $\mathcal{A} = (\mathcal{I}_\mathcal{A}, \mathcal{R}_\mathcal{A})$ where $\mathcal{I}_\mathcal{A}$ is a probability distribution over the possible values of the secret, encoding the attacker's prior information about it, and $\mathcal{R}_\mathcal{A}$ is a discrimination relation over the states of the system in which two states are in the same class iff the attacker cannot discriminate them.

The attacker has some prior information about the secret before observing the process. This is encoded as the prior $\mathcal{I}_\mathcal{A}$, that assigns probability distributions to all high-level variables. In particular, since we assume that the attacker has access to the source code of the process, he will know the length in bits of the secret variables, since he can read the variable declarations. For instance if the secret is declared as a 16-bit integer value the attacker will know that it is an integer from 0 to 65535. Any additional information the attacker possesses, like that the secret is not a number from 18 to 556, or that the probability that the secret is 12 is three times the probability that it is 16, is encoded in the prior distribution as well.

The discrimination relation $\mathcal{R}_\mathcal{A}$ encodes what states of the MDP the attacker can discriminate. The usual case is that some of the low-level variables are observable by the attacker, and thus the attacker can discriminate two states iff they differ in the value of at least one of the observable variables. States with the same values for all the observable variables are indistinguishable to the attacker.

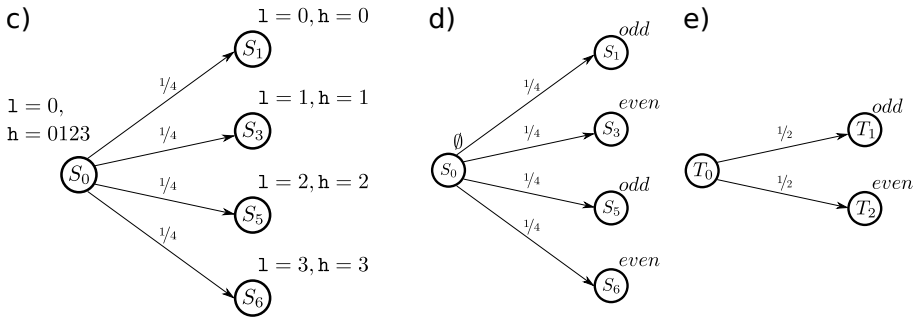

Figure 4: Simple loop example c) Observable reduction d) Relabeling e) Quotient

Example. In our example we will use the following attacker: $\mathcal{I}_{\mathcal{A}} = (1/4, 1/4, 1/4, 1/4)$ (no prior information) and $\mathcal{R}_{\mathcal{A}} = \{(S_1, S_5), (S_3, S_6)\}$ (cannot distinguish states S_1 from S_5 and S_3 from S_6).

Step 3: Resolve the nondeterminism in the MDP. To transform the MDP in a MC, in order to compute leakage, we need to exploit the prior information $\mathcal{I}_{\mathcal{A}}$ of the attacker. We use it to compute a probability distribution over possible values of private variables in each states of the MDP. To do this for a given state s we just need to normalize $\mathcal{I}_{\mathcal{A}}$ on the allowed values of the private variables for the state. The probability of the each action $a \in \Lambda_s$ is computed as the probability of the event labeling a given the probability distribution over the values of the secret in s. We will denote the obtained MC by $\mathcal{C}$.

Example. In state S_0 the probability distribution over `h` is $\mathcal{I}_{\mathcal{A}} = (1/4, 1/4, 1/4, 1/4)$ and `l=0`. The program transitions to state S_1 if `h=l` and to state S_2 if `h`$\neq$`l`. We have that P_{S_0,S_1} is $\mathbf{P}(\mathtt{h} = \mathtt{l}|S_0) = 1/4$ and the probability distribution on `h` in S_1 is $(1, 0, 0, 0)$. Complementarily, P_{S_0,S_2} is $3/4$ and the probability distribution on `h` in S_2 is $(0, 1/3, 1/3, 1/3)$. Figure 3b shows the Markov chain $\mathcal{C}$ obtained by repeating this step in all states of the MDP of Fig. 3a.

Step 4: Hide internal states (Sect. 5). We want to model the fact that most of the states of the system are not observable to the attacker. We assume that the attacker can start the process and then only observe the value of the output after the process terminates, so all states except the initial state and the absorbing states must be hidden. Let $\mathcal{T}$ be the set of all states of the Markov chain except the initial state and the absorbing states.

The procedure is implemented in Algorithm 1 in Section 5. We call $\mathbb{C}$ the *observable reduction* of the scenario. Importantly, $\mathbb{C}$ is a one-step Markov chain.

Example. Figure 4c presents the observable reduction for the running example.

Step 5: Compute the Quotients. Having modeled the problem scenario as a one-step Markov chain, we now want to compute the information leakage of the scenario. To compute it we apply the formula for mutual information $I(O, h) = H(O) + H(h) - H(O, h)$ introduced in Section 2, where (O, h) is the

joint distribution of the secret and observable variables and (O) and (h) are the marginal distributions. At this step all three will be modeled as one-step Markov chains, respectively $\mathbb{C}_{O,h}$, $\mathbb{C}_O$ and $\mathbb{C}_h$. We need to produce the three Markov chains and compute their entropy.

The three Markov chains are obtained from the observable reduction by quotienting it by appropriate discrimination relations, as explained in Definition 3. The discrimination relation to obtain $\mathbb{C}_O$ is $\mathcal{R}_\mathcal{A}$, as it represents the attacker's discrimination on the secret. We will call $\mathbb{C}_O = \mathbb{C}/\mathcal{R}_\mathcal{A}$ the *attacker's quotient*.

Since $\mathbb{C}$ is a one-step Markov chain, the only transitions are the ones from the starting states to the absorbing states and the looping transitions with probability 1 of the absorbing states, so the quotient collapses together absorbing states in the same equivalence class. The transition probability from the starting state to a class is the sum of the transition probabilities from the starting state to the states composing the class.

The *secret's quotient* $\mathbb{C}_h$ represents the marginal distribution of the secret variable. Its entropy is also a measure of how many bits of the secret variable are actually used by the process. Let $\mathcal{R}_h$ be a discrimination relation such that two states are in the same class if and only if they assign the same set of possible values to the secret variables. Then $\mathbb{C}_h = \mathbb{C}/\mathcal{R}_h$.

Finally we compute the *joint quotient* $\mathbb{C}_{O,h}$ representing the joint behavior of the secret and observable variables. It is obtained by quotienting the scenario Markov chain model by $\mathcal{R}_\mathcal{A} \cap \mathcal{R}_h$, thus lumping in the same class only states that agree both on the values of the observable variables and on the set of possible values for the secret variables: $\mathbb{C}_{O,h} = \mathbb{C}/(\mathcal{R}_\mathcal{A} \cap \mathcal{R}_h)$.

Example. Recall that in the running example the attacker is only able to read the parity of `l`. We have that $\mathcal{R}_\mathcal{A} = \{(S_1, S_5), (S_3, S_6)\}$. We name the equivalence classes *even* and *odd* and relabel the state with the classes (see Fig. 4d). The attacker's quotient for the running example is shown in Fig. 4e. Since this example is deterministic it is not necessary to compute the secret's quotient and the joint quotient, as explained in the next paragraph.

Step 6: Compute the Leakage. Finally, we compute the information leakage by computing $H(\mathbb{C}_O)$, $H(\mathbb{C}_h)$ and $H(\mathbb{C}_{O,h})$ and applying the formula

$$I(O, h) = H(\mathbb{C}_O) + H(\mathbb{C}_h) - H(\mathbb{C}_{O,h})$$

This requires us to be able to compute the entropy of Markov chains. Since in this procedure the chains are reduced by the hiding algorithm to a single step it reduces to computing the local entropy of the initial state of each chain.

Note that if a process is deterministic, i.e. never uses the random assignment function, then all uncertainty in the system follows from lack of information about the secret, so $H(\mathcal{C}_h) = H(\mathcal{C}_{O,h})$ and consequently $I(O, h) = H(\mathcal{C}_O)$. In this case we can simply compute the attacker's quotient and its entropy, as it corresponds with the information leakage. This will be explained in more details in Section 6.

Example. The program is deterministic, the leakage of the scenario $(\mathcal{P}, \mathcal{A})$ is equivalent to the entropy of the attacker's quotient in Fig. 4e, or 1 bit.

```
...
42 x := x +3;
43 if (x<8) then x := x + 1;
...
```

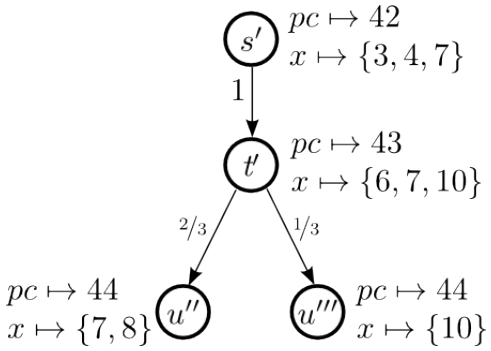

Figure 5: Markov chain semantics for a snippet of imperative code

4. Handling Randomized Imperative Programs

A Markov chain is a probabilistic process respecting the Markov property, meaning that the distribution over the states at a certain time depends only on the distribution at the time immediately before. This means that in each state of the Markov chain there must be enough information to determine the probability distribution on the next step. The state of the Markov chain captures the program state, so valuations of all variables and the program counter.

4.1. A Simple Example

Let line 42 in the source code be `x := x + 3;`, and let state s be the assignment $\{pc \mapsto 42, x \mapsto 3\}$, where pc is the value of the program counter. Then in the model there will be a transition with probability 1 to a state t where $\{pc \mapsto 43, x \mapsto 6\}$: the program counter is increased by 1 as a default behavior, and the variable x is increased by 3 according to the instruction we are processing.

Since imperative programming languages are deterministic, for each state there will be exactly one other state that will follow with probability 1. This would lead to a trivial and uninteresting Markov chain. The reason why we are using probabilistic processes as models is their capability of representing imperfect information about the behavior of a system. We use this to represent what an attacker with imperfect information about the process knows about the process' behavior. In practice we lift the states to represent not just an assignment of values to the variables, but a level of information of an attacker about these assignments.

Imagine that we allow states to represent assignment of sets of values to each variable except the program counter. Then for the example above we may have a state $s' = \{pc \mapsto 42, x \mapsto \{3, 4, 7\}\}$ representing the fact that the variable

x could have value either 3, 4 or 7. From s' we would have a transition with probability 1 to state $t' = \{pc \mapsto 43, x \mapsto \{6, 7, 10\}\}$, representing the fact that variable x has been increased by 3 and is now either 6, 7 or 10.

Now imagine that line 43 of the source code is `if (x<8) then x := x + 1;`. Then from state t' we could only have a transition with probability 1 to state $u' = \{pc \mapsto 44, x \mapsto \{7, 8, 10\}\}$. But we could actually define two more precise successor states for t': $u'' = \{pc \mapsto 44, x \mapsto \{7, 8\}\}$ is visited if the guard is evaluated to be true and $u''' = \{pc \mapsto 44, x \mapsto \{10\}\}$ is visited if the guard is evaluated to false.

The probability of transitioning from t' to u'' and u''' is the probability that the guard is true or false in t', respectively. This means that if we have in each state a probability distribution over possible values of each variable we can have two possible successor states each time a conditional statement is evaluated, and in the state we have enough information to compute the probability that the guard will be true or false. If in t' we have that variable x has a uniform distribution over the three values 6, 7 and 10 then we can compute that the guard `x<8` will be true with probability $2/3$ and false with probability $1/3$, thus the probability of transitioning from t' to u'' is $2/3$ and the probability of transitioning from t' to u''' is $1/3$. The snippet of code and resulting Markov chain are depicted in Fig. 5.

Note that, since we assume a distribution on the variable x in every state of the Markov chain, we could have split t' or even s' in three different states, one for each possible assignment of values to x, and given each of them a starting probability depending on the initial distribution over x. This would have meant encoding a different level of information about the attacker, and would also have multiplied the number of states of the chain. The level of abstraction we apply in this work represents the fact that the attacker knows the bit size of the secret variables but not their exact value.

4.2. Semantics

We give a simple probabilistic partial-observation semantics for an imperative language with randomization. This semantics, akin to abstract interpretation, derives Markovian models of finite state programs automatically.

We distinguish a list of variables (pc, L, H) where pc is the program counter, L is the set of public variables and H is the set of private variables. The secret is one of the private variables. In each state a given value is assigned to each variable in L and to the program counter, while a probability distribution over a set of possible values is assigned to each variable in H. Assume that all variables are integer of fixed size. While variables can be declared at any point of the code, we assume that no two variables are declared with the same name, so a global variable scope is sufficient. High-level variables are read-only and cannot be accessed externally; their value estimation changes only through observation via evaluation of branch conditions. This models the fact that the attacker cannot directly print or modify a secret value, but only learn about it by observing the decisions that the process takes that depend on it.

$$\frac{pc\text{: } \texttt{public int}n\ \texttt{v} := k}{(pc, L, H) \xrightarrow{\top} [1 \mapsto (pc+1, L \cup \{(\texttt{k}/\texttt{v}, n)\}, H)]}$$

$$\frac{pc\text{: } \texttt{private int}n\ \texttt{h}}{(pc, L, H) \xrightarrow{\top} [1 \mapsto (pc+1, L, H \cup \{(\{\texttt{0},\ldots,2^n\texttt{-1}\}/\texttt{h}, n)\})]}$$

$$\frac{pc\text{: } \texttt{skip}}{(pc, L, H) \xrightarrow{\top} [1 \mapsto (pc+1, L, H)]} \qquad \frac{pc\text{: } \texttt{v} := \texttt{f(l)}}{(pc, L, H) \xrightarrow{\top} [1 \mapsto (pc+1, L[\texttt{f(l)}/\texttt{v}], H)]}$$

$$\frac{pc\text{: } \texttt{v := rand } p}{(pc, L, H) \xrightarrow{\top} [p \mapsto (pc+1, L[\texttt{0}/\texttt{v}], H), (1-p) \mapsto (pc+1, L[\texttt{1}/\texttt{v}], H)]}$$

$$\frac{pc\text{: } \texttt{goto label}}{(pc, L, H) \xrightarrow{\top} [1 \mapsto (\texttt{label}, L, H)]} \qquad \frac{pc\text{: } \texttt{return}}{(pc, L, H) \xrightarrow{\top} [1 \mapsto (pc, L, H)]}$$

$$\frac{pc\text{: } \texttt{if g(l,h) then la: A else lb: B}}{(pc, L, H) \xrightarrow{\texttt{g(l,h)}} [1 \mapsto (\texttt{la}, L, H|\texttt{g(l,h)})]}$$

$$\frac{pc\text{: } \texttt{if g(l,h) then la: A else lb: B}}{(pc, L, H) \xrightarrow{\neg\texttt{g(l,h)}} [1 \mapsto (\texttt{lb}, L, H|\neg\texttt{g(l,h)})]}$$

Figure 6: Execution rules in probabilistic partial information semantics generating a MDP.

Let l, v (resp. h) range over names of public (resp. private) variables and p range over reals from $[0; 1]$. Let `label` denote program points and `f` (`g`) pure arithmetic (Boolean) expressions. Assume a standard set of expressions and the following statements:

$$\begin{aligned} \textit{stmt} ::= \ & \texttt{public int}n\ \texttt{v} \mid \texttt{private int}n\ \texttt{h} \mid \texttt{v := f(}l\ldots\texttt{)} \mid \texttt{v := rand } p \mid \\ & \texttt{skip} \mid \texttt{goto label} \mid \texttt{return} \mid \texttt{if g(}l\ldots\texttt{,}h\ldots\texttt{)} \ \ \texttt{then } \textit{stmt-list} \\ & \texttt{else } \textit{stmt-list} \end{aligned}$$

The first statement declares a public variable `v` of size n bits with a given value k, while the second statement similarly declares a private variable `h` of size n bits with allowed values ranging from 0 to $2^n - 1$. Remember that we assume that there exists and arbitrarily large integer M such that variables larger than M cannot be declared, so in both declarations $n \leq M$. The third statement assigns to a public variable the value of expression `f` depending on other public variables. The fourth statement assigns zero with probability p, and one with probability $1-p$, to a public variable. The `return` statement outputs values of all public variables and terminates. A conditional branch first evaluates an expression `g` dependent on private and public variables; the first list of statements is executed if the condition holds, and the second otherwise. For

simplicity, all statement lists must end with an explicit jump, as in: `if g(l,h) then ...; goto done; else ...; goto done; done:` ... . Each program can be easily transformed to this form. Since only a single variable scope exists, loops can be added in a standard way as a syntactic sugar.

The semantics (Fig. 6) is a small-step operational semantics with transitions from states to distributions over states, labeled by expressions dependent on h (only used for the conditional statement). It generates an MDP over the reachable state space. In Fig. 6, `v`, `l` are public variables and `h` is a private variable. Expressions in rule consequences stand for values obtain in a standard way. $L[^X/_l]$ denotes substitution of X as the new value for l in mapping L. Finally, $H|\mathtt{g}$ denotes a restriction of each set of possible values in a mapping H, to contain only values that are consistent with Boolean expression `g`. Observe that the `return` rule produces an absorbing state—this is how we model termination in an MDP. The `rand` rules produces a proper distribution, unlike the Dirac distributions. The `if` rule produces a nondeterministic decision state.

In the obtained MDP states are labeled by values of public variables and sets of values of private variables. Actions from each state represent the secret-dependent events for the state. Also, the set of allowed values for each secret variable in the successor of a given state is be a subset of the allowed values for the same variable in the state itself. Our leakage quantification technique works for any MDP with these properties, even the ones not necessarily obtained from code. In Sect. 9 we will create such a model directly from a topology of the Onion Routing protocol.

5. Hiding Internal States

In the simple loop example of Sect. 3 the attacker is unable to observe states S_2 and S_4; we call these non-observable states *hidden*. His view of the system is thus adequately represented by the MC in Fig. 4c. In this figure the probability of going from the state S_0 to state S_5 is the probability of reaching S_5 from S_0 in the MC of Fig. 3b *eventually*, after visiting zero or more unobservable intermediate states. These states can be eliminated (hidden) for the purpose of leakage computation.

Hiding the internal states models the fact that the attacker can observe the values of the observable variables after the termination of the system. If the system does not terminate, the attacker knows that the system did not terminate but can read no value of the observable variables. This modeling choice is consistent with the assumptions behind the channel matrix model.

Note that the initial state cannot be hidden, as we assume the attacker knows that the system is running. This assumption does not restrict the power of the approach, since one can always model a system, whose running state is unknown, by prefixing its initial state by a pre-start state, making it initial, and hiding the original initial state.

The distinction between the indistinguishable states encoded in $\mathcal{R}_{\mathcal{A}}$ and the hidden states encoded in $\mathcal{T}$ is fine but fundamental. While the attacker

Data: A Markov chain $\mathcal{C} = (S, s_0, P)$, the set $\mathcal{T} \subseteq S$ of states to hide.
Result: The Markov chain $\mathcal{C} = (S \cup \{\uparrow\} \setminus \mathcal{T}, s_0, P)$
Add to S the divergence state $\uparrow$ with $P_{\uparrow,\uparrow} = 1$ and $\pi^{(0)}_{\uparrow} = 0$;
while $T \neq \emptyset$ **do**
  Choose a state $t \in \mathcal{T}$;
  if $P_{t,t} = 1$ **then**
    $\pi^{(0)}_{\uparrow} \leftarrow \pi^{(0)}_{\uparrow} + \pi^{(0)}_{t}$
    foreach $s \in S$ **do**
      $P_{s,\uparrow} \leftarrow P_{s,\uparrow} + P_{s,t}$
      $P_{s,t} \leftarrow 0$
    end
  else
    foreach $u \in S$ **do**
      $P^{\square}_{t,u} \leftarrow \frac{P_{t,u}}{1-P_{t,t}}$
      $\pi^{(0)}_{u} \leftarrow \pi^{(0)}_{u} + \pi^{(0)}_{t} P^{\square}_{t,u}$
      foreach $s \in S$ **do**
        $P_{s,u} \leftarrow P_{s,u} + P_{s,t} P^{\square}_{t,u}$
        $P_{s,t} \leftarrow 0$
      end
    end
  end
  $S \leftarrow S \setminus \{t\}$
  $\mathcal{T} \leftarrow \mathcal{T} \setminus \{t\}$
end

Algorithm 1: Hide a subset of the states of a Markov chain.

can observe whether the system is in one of the equivalence classes of the discrimination relation $\mathcal{R}_{\mathcal{A}}$, he is not even aware of the existence of the hidden states in the set $\mathcal{T}$. For instance this means that he can count how many time steps the chain spends in an equivalence class of $\mathcal{R}_{\mathcal{A}}$ but not how many steps the chain spends in the hidden states. To encode the attacker that only observes the output of the process after its termination, we hide all states except the starting state and the absorbing states of the chain: the attacker knows when the process starts and ends but cannot perform observations about what happens during the computation.

The fact that the hidden states in $\mathcal{T}$ cannot be observed raises a question: what happens if the process terminates in one of these states? In this case the attacker would not be able to observe the termination of the program. Similarly, what if the process does not terminate at all, and its behavior loops forever within these hidden states? It can be seen that the two cases are equivalent from the point of view of the attacker: he observes that the program gets to a certain point and that nothing else happens, including termination. To address this problem we consider termination as one of the possible observable outputs of

the program and introduce a nontermination absorbing state modeling this case.

Algorithm 1 details the procedure. We will overload the set difference symbol $\setminus$ to use for the hiding operation: we write $\mathbb{C} = \mathcal{C} \setminus \mathcal{T}$ for the observable MC obtained from $\mathcal{C}$ by hiding the states in set $\mathcal{T}$. If a system stays in a hidden state forever, we say it *diverges*. Divergence will be symbolized by a dedicated absorbing state named $\uparrow$.

The hidden states are removed from the chain one by one, and each time the transition probability of the other states are modified to consider the probability of eventually transitioning from one state to the other through the hidden state. The initial probability $\pi^{(0)}$ is similarly updated.

6. Computing Quotients and Leakage

At this point we have the observable reduction $\mathbb{C}$ obtained through the hiding protocol. Remember that $\mathbb{C}$ is a one-step Markov chain.

We want to compute the three quotients introduced in Step 5 of Section 3:

- The attacker's quotient $\mathbb{C}_O = \mathbb{C}/\mathcal{R}_\mathcal{A}$, representing the distribution over the observables;
- The secret's quotient $\mathbb{C}_h = \mathbb{C}/\mathcal{R}_h$ with $\mathcal{R}_h = \{(s,t) \in S \times S \mid \mathtt{h}(s) = \mathtt{h}(t)\}$ representing the distribution over the values of the secret variable $\mathtt{h}$;
- The joint quotient $\mathbb{C}_{O,h} = \mathbb{C}/(\mathcal{R}_\mathcal{A} \cap \mathcal{R}_h)$ representing the joint distribution over values of the secret and observables.

The next definition applies the mutual information formula of Definition 4 to compute the Shannon leakage of a scenario:

Definition 9. Let $(\mathcal{P}, \mathcal{A})$ be a scenario, $\mathcal{A} = (\mathcal{I}_\mathcal{A}, \mathcal{R}_\mathcal{A})$ an attacker, and $\mathbb{C}_o$, $\mathbb{C}_h$ and $\mathbb{C}_{O,h}$ the attacker's, secret's and joint quotient, respectively. Then the information leakage of $\mathcal{P}$ to $\mathcal{A}$ is

$$\mathit{Leakage}(\mathcal{P}, \mathcal{A}) = I(\mathbb{C}_O; \mathbb{C}_h) = H(\mathbb{C}_O) + H(\mathbb{C}_h) - H(\mathbb{C}_{O,h}).$$

Corollary 1. *If $\mathcal{P}$ is a deterministic program, then the leakage is $H(\mathbb{C}_O)$.*

The common definition of leakage of the `LoI` approach [2] assumes that the attacker can observe the different output of a deterministic system. It can be easily encoded in our method. Consider a deterministic program $\mathcal{P}$ with a low-level variable $\mathtt{o}$ encoding the output of the program, $H(\mathit{LoI}(\mathcal{P}))$ as defined in [2], and $\mathit{Leakage}(\mathcal{P}, \mathcal{A})$ as defined in Definition 9. Let the an attacker $\mathcal{A}_{I/O}$ have $\mathcal{R}_{\mathcal{A}_{I/O}} = \{(s,t) \in S \times S \mid \mathtt{o}(s) = \mathtt{o}(t)\}$. The following proposition states that the attacker $\mathcal{A}_{I/O}$ is the one considered in [2]:

Theorem 1. *Let $(\mathcal{P}, \mathcal{A}_{I/O})$ be a scenario, with $\mathcal{P}$ deterministic and $\mathcal{A}_{I/O}$ being the attacker defined above. Then*

$$H(\mathbb{C}_O) = H(\mathit{LoI}(\mathcal{P}))$$

The proof of Theorem 1 is based on the following Corollary:

Corollary 2. *Let $\mathcal{P}$ be a program and o_i its outputs. For each o_i, the total probability of reaching absorbing states labeled with o_i in the MDP semantics of $\mathcal{P}$ is equal to $\mathbf{P}([[\mathcal{P}]] = o_i)$ where $[[\mathcal{P}]]$ is the r.v. derived from the denotational semantics of the program (i.e. LoI($\mathcal{P}$)).*

Proof. □

Now we can proceed to the proof of Theorem 1:

Proof of Theorem 1. Note that since all internal states are hidden, the MC is a 1-step probability distribution from the starting state to the output states. By Corollary 2 the probability of observing the observations o_i is consistent with the probability of observing the same observations in $\mathcal{P}$, thus $H(\mathcal{C}_O) = H(\mathcal{C}/\mathcal{R}_{\mathcal{A}_{\mathcal{I}/\mathcal{O}}}) = H([[\mathcal{P}]]) = H(LoI(\mathcal{P}))$. □

7. Leakage Ordering for Probabilistic Programs

Discrimination relations are equivalence relations in which an equivalence class represents a set of states that cannot be distinguished by the attacker. Different attackers have different discriminating power. We call $\mathcal{R}_{\mathcal{A}}$ the discrimination relation of an attacker. A discrimination relation is encoded in the program by quotienting the observable reduction $\mathbb{C}$ by $\mathcal{R}_{\mathcal{A}}$

This procedure generalizes the discrimination relation ordering used in the `LoI` approach [1] and allows us to define a leakage ordering that is also valid for probabilistic programs. Let $\mathcal{A}_1 = (\mathcal{I}_{\mathcal{A}}, \mathcal{R}_{\mathcal{A}_1})$ and $\mathcal{A}_2 = (\mathcal{I}_{\mathcal{A}}, \mathcal{R}_{\mathcal{A}_2})$ be two attackers sharing the same prior information, and define

$$\mathcal{A}_1 \sqsubseteq \mathcal{A}_2 \quad \text{iff} \quad \mathcal{R}_{\mathcal{A}_1} \supseteq \mathcal{R}_{\mathcal{A}_2}$$

Theorem 2. *Let $\mathcal{A}_1$ and $\mathcal{A}_2$ be two attackers such that $\mathcal{A}_1 \sqsubseteq \mathcal{A}_2$. Then for any program $\mathcal{P}$, the leakage of the scenario $(\mathcal{P}, \mathcal{A}_1)$ is greater or equal then the leakage of the scenario $(\mathcal{P}, \mathcal{A}_2)$.*

Proof. The theorem can be considered a consequence of a known result by Nakamura [27, Section 4.3] Let X be the random variable on the space $\mathcal{X}$ representing the value of the secret. For $j \in \{1, 2\}$ let O_j be the random variable on the space $\mathcal{O}_j$ representing the observable value observed by the attacker $\mathcal{A}_j$. Then the leakage of the scenario $(\mathcal{P}, \mathcal{A}_j)$ is $I(X; O_j) = H(X) - H(X|O_j)$. Since $\mathcal{A}_1 \sqsubseteq \mathcal{A}_2$, then the outcomes in $\mathcal{O}_1$ partition the outcomes in $\mathcal{O}_2$ such that for each outcome $o_1 \in \mathcal{O}_1$ there exists a nonempty set of outcomes $Q_{o_1} = \{o_{21}, ..., o_{2k}\} \subseteq \mathcal{O}_2$ satisfying $P(o_1) = P(o_{21}) + ... + P(o_{2k}) = P(Q_{o_1})$. The

claim of the theorem can be rewritten as

$$
\begin{aligned}
& H(X) - H(X|O_1) \leq H(X) - H(X|O_2) \\
\equiv\ & H(X|O_1) \geq H(X|O_2) \\
\equiv\ & -\sum_{o_1 \in \mathcal{O}_1} P(o_1) H(X|O_1 = o_1) \geq -\sum_{o_2 \in \mathcal{O}_2} P(o_2) H(X|O_2 = o_2) \\
\equiv\ & \sum_{o_1 \in \mathcal{O}_1} P(o_1) \sum_{x \in \mathcal{X}} P(x|o_1) \log P(x|o_1) \leq \sum_{o_2 \in \mathcal{O}_2} P(o_2) \sum_{x \in \mathcal{X}} P(x|o_2) \log P(x|o_2) \\
\equiv\ & \sum_{o_1 \in \mathcal{O}_1} \sum_{x \in \mathcal{X}} P(x, o_1) \log \frac{P(x, o_1)}{P(o_1)} \leq \sum_{o_2 \in \mathcal{O}_2} \sum_{x \in \mathcal{X}} P(x, o_2) \log \frac{P(x, o_2)}{P(o_2)} \\
\equiv\ & \sum_{o_1 \in \mathcal{O}_1} \sum_{x \in \mathcal{X}} P(x, o_1) \log \frac{P(x, o_1)}{P(o_1)} \leq \sum_{o_1 \in \mathcal{O}_1} \sum_{o_2 \in Q_{o_1}} \sum_{x \in \mathcal{X}} P(x, o_2) \log \frac{P(x, o_2)}{P(o_2)} \\
\Leftarrow\ & \forall_{x \in \mathcal{X}}. \forall_{o_1 \in \mathcal{O}_1}. \left(P(x, o_1) \log \frac{P(x, o_1)}{P(o_1)} \leq \sum_{o_2 \in Q_{o_1}} P(x, o_2) \log \frac{P(x, o_2)}{P(o_2)} \right) \\
& \qquad \text{(since the Q sets partition } \mathcal{O}_2) \\
\equiv\ & \forall_{x \in \mathcal{X}}. \forall_{o_1 \in \mathcal{O}_1}. \left(P(x, Q_{o_1}) \log \frac{P(x, Q_{o_1})}{P(Q_{o_1})} \leq \sum_{o_2 \in Q_{o_1}} P(x, o_2) \log \frac{P(x, o_2)}{P(o_2)} \right) \\
& \qquad \text{(since } P(o_1) = P(Q_{o_1})) \\
\equiv\ & \forall_{x \in \mathcal{X}}. \forall_{o_1 \in \mathcal{O}_1}. \left(\sum_{o_2 \in Q_{o_1}} P(x, o_2) \log \frac{P(x, Q_{o_1})}{P(Q_{o_1})} \leq \sum_{o_2 \in Q_{o_1}} P(x, o_2) \log \frac{P(x, o_2)}{P(o_2)} \right) \\
& \qquad \text{(since } P(x, Q_{o_1}) = P(x \wedge (o_{21} \vee ... \vee o_{2k}))))
\end{aligned}
$$

which holds by the log sum inequality. □

Effectively, the attacker that is able to discriminate more states (a language-like qualitative property) is able to get more information by observing the system (an information- theoretical quantitative property). The attacker who can get most information is the one who can discriminate all states, thus its discrimination relation is the identity. The attacker getting the least information is the one who cannot discriminate any state from any other: to this attacker the system leaks no information.

8. Computing Channel Capacity

The method we presented computes the leakage for a scenario, but it is common in security to ask what is the leakage of a given program in the worst-case scenario, i.e. for the scenario with the highest leakage. We consider the maximum leakage over all the attackers with the same discrimination relation

$\mathcal{R}_\mathcal{A}$ but different prior information $\mathcal{I}_\mathcal{A}$. We define a class of attackers this way because maximizing over all discrimination relations would just conclude that the attacker able to discriminate all states leaks all the information in the system. The maximum leakage for a class of attackers is known as channel capacity, and it is the upper bound to the leakage of the system to any attacker [8]:

Definition 10. Let $\mathcal{P}$ be a program and $\mathbb{A}$ the class of all attackers with discrimination relation $\mathcal{R}_\mathbb{A}$. Let $\hat{\mathcal{A}} \in \mathbb{A}$ be the attacker maximizing the leakage of the scenario $(\mathcal{P}, \mathcal{A})$ for all $\mathcal{A} \in \mathbb{A}$. Then the channel capacity of $\mathcal{P}$ is the leakage of the scenario $(\mathcal{P}, \hat{\mathcal{A}})$.

To compute it we proceed as follows. We first transform the MDP semantics of $\mathcal{P}$ in a parametrized MC with constraints. Then we define a MC and a reward function from it such that the expected total reward of the MC is equivalent to the leakage of the system. Then we extract an equation with constraints characterizing this reward as a function of the prior information $\mathcal{I}_\mathcal{A}$ of the attacker. Finally, we maximize the equation and obtain the maximum leakage, i.e. the channel capacity. In the next Section we will apply this method to compute the channel capacity of attacks to the Onion Routing protocol.

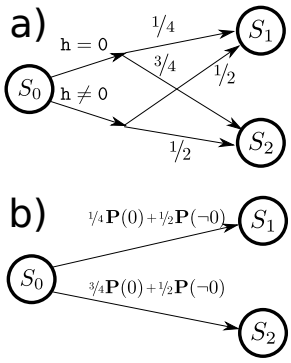

Figure 7: Reduction from MDP to parameterized MC

Step 1: Find the parametrized MC. We abuse the notation of Markov chain allowing the use of variables in the transition probabilities. This allows us to transform the MDP semantics of a program $\mathcal{P}$ in a MC with the transition probabilities parametrized by the probability of choosing the actions in each state.

Consider the MDP in Fig 7a; in state S_0 either $\mathtt{h} = \mathtt{0}$ or $\mathtt{h} \neq \mathtt{0}$ and the system moves to the next state with the appropriate transition probability. Let $\mathbf{P}(0)$ and $\mathbf{P}(\neg 0)$ be $\mathbf{P}(\mathtt{h} = \mathtt{0}|S_0)$ and $\mathbf{P}(\mathtt{h} \neq \mathtt{0}|S_0)$ respectively; then we can transform the MDP in the MC in Fig 7b, with the constraint $\mathbf{P}(0) + \mathbf{P}(\neg 0) = 1$.

We hide the internal states in the MC obtaining the one-step observational reduction $\mathbb{C}$, as described in Sect. 5.

Step 2: Define a reward function for leakage. We want to define a reward function on the parametrized MC such that the expected total reward of the chain is equivalent to the leakage of the system. This step can be skipped if the leakage equation can be obtained directly from the model, like in the examples in the next Section. In the example in Fig. 7 the system is deterministic, so its

leakage is equal to its entropy by Corollary 1, and we just need to define the entropy reward function on transitions $R(s,t) = -\log_2 P_{s,t}$, as explained in [15].

For a probabilistic system we need to build another MC by composing the three quotients $\mathbb{C}_O$, $\mathbb{C}_h$ and $\mathbb{C}_{O,h}$, and we define the leakage reward function on the composed chain:

Theorem 3. *Let $\mathbb{C}$ be the parallel composition of $\mathbb{C}_O$, $\mathbb{C}_h$ and $\mathbb{C}_{O,h}$. Let R be a reward function on the transitions of $\mathbb{C}$ such that*

$$R(s_1 \times s_2 \times s_3, t_1 \times t_2 \times t_3) = \log_2 \frac{P_{s_1,t_1} P_{s_2,t_2}}{P_{s_3,t_3}}.$$

Then the expected total infinite time reward of $\mathbb{C}$ with the reward function R is equivalent to $H(\mathbb{C}_O) + H(\mathbb{C}_h) - H(\mathbb{C}_{O,h})$ and thus to the leakage.

The proof of Theorem 3 follows immediately from the following Lemma:

Lemma 2. *Let $\mathcal{C}_1 = (S_1, s_0^1, P_1)$, $\mathcal{C}_2 = (S_2, s_0^2, P_2)$ be Markov chains. Let $\mathcal{C}(S, s_0, P)$ be their synchronous parallel composition, i.e. $S = S_1 \times S_2$, $s_0 = s_0^1 \times s_0^2$ and*

$$P_{s_1 \times s_2, t_1 \times t_2} = P_{s_1,t_1} P_{s_2,t_2}.$$

Let R^+ and R^- be reward functions on the transitions of $\mathcal{C}$ such that

$$R^+(s_1 \times s_2, t_1 \times t_2) = \log_2 \left(P_{s_1,t_1} P_{s_2,t_2}\right).$$

$$R^-(s_1 \times s_2, t_1 \times t_2) = \log_2 \left(\frac{P_{s_1,t_1}}{P_{s_2,t_2}}\right).$$

Then the expected total infinite time reward of $\mathcal{C}$ with the reward function R^+ is equivalent to $H(\mathcal{C}_1) + H(\mathcal{C}_2)$ and the expected total infinite time reward of $\mathcal{C}$ with the reward function R^- is equivalent to $H(\mathcal{C}_1) - H(\mathcal{C}_2)$.

Proof. We will prove the result for $\mathcal{R}^-$; the proof for $\mathcal{R}^+$ is symmetrical. Consider a state $s = s_1 \times s_2$ of $\mathcal{C}$. The expected reward of s is

$$\begin{aligned}
R^-(s) &= \sum_{t \in S} P_{s,t} R^-(s,t) \\
&= \sum_{t_1 \in S_1} \sum_{t_2 \in S_2} P_{s_1,t_1} P_{s_2,t_2} \log_2 \frac{P_{s_1,t_1}}{P_{s_2,t_2}} \\
&= \sum_{t_1 \in S_1} \sum_{t_2 \in S_2} P_{s_1,t_1} P_{s_2,t_2} (\log_2 P_{s_1,t_1} - log_2 P_{s_2,t_2}) \\
&= \sum_{t_1 \in S_1} \sum_{t_2 \in S_2} P_{s_1,t_1} P_{s_2,t_2} \log_2 P_{s_1,t_1} - \sum_{t_1 \in S_1} \sum_{t_2 \in S_2} P_{s_1,t_1} P_{s_2,t_2} \log_2 P_{s_2,t_2} \\
&= \sum_{t_2 \in S_2} P_{s_2,t_2} \sum_{t_1 \in S_1} P_{s_1,t_1} \log_2 P_{s_1,t_1} - \sum_{t_1 \in S_1} P_{s_1,t_1} \sum_{t_2 \in S_2} P_{s_2,t_2} \log_2 P_{s_2,t_2} \\
&= 1 \cdot \sum_{t_1 \in S_1} P_{s_1,t_1} \log_2 P_{s_1,t_1} - 1 \cdot \sum_{t_2 \in S_2} P_{s_2,t_2} \log_2 P_{s_2,t_2} \\
&= L(s_1) - L(s_2)
\end{aligned}$$

thus the expected total reward of $\mathcal{C}$ is

$$\begin{aligned}
R^-(C) &= \sum_{s\in S} R^-(s)\xi_s \\
&= \sum_{s\in S} R^-(s) \sum_{n=0}^{\infty} P^n_{s_0,s} \\
&= \sum_{s_1\in S_1}\sum_{s_2\in S_2} (L(s_1)-L(s_2)) \sum_{n=0}^{\infty} P^n_{s_0^1,s_1} P^n_{s_0^2,s_2} \\
&= \sum_{s_1\in S_1} L(s_1)\left(\sum_{n=0}^{\infty} P^n_{s_0^1,s_1} \sum_{s_2\in S_2} P^n_{s_0^2,s_2}\right) - \sum_{s_2\in S_2} L(s_2)\left(\sum_{n=0}^{\infty}(P^n_{s_0^2,s_2} \sum_{s_1\in S_1} P^n_{s_0^1,s_1}\right) \\
&= \sum_{s_1\in S_1} L(s_1) \sum_{n=0}^{\infty}\left(P^n_{s_0^1,s_1}\cdot 1\right) - \sum_{s_2\in S_2} L(s_2)\sum_{n=0}^{\infty}\left(P^n_{s_0^2,s_2}\cdot 1\right) \\
&= \sum_{s_1\in S_1} L(s_1)\xi_{s_1} - \sum_{s_2\in S_2} L(s_2)\xi_{s_2} \\
&= H(\mathcal{C}_1) - H(\mathcal{C}_2).
\end{aligned}$$

□

Step 3: Extract the leakage as an equation. Now that we have a reward function R on the transitions of a MC characterizing the leakage of the system, we need to maximize it. One possible strategy is to extract the explicit equation of the reward of the chain as a function of the transition probabilities, which themselves are a function of the prior information $\mathcal{I}_{\mathcal{A}}$. For a reward function $R(s,t)$ on transitions the reward for the MC is

$$R(\mathcal{C}) = \sum_{s\in S} R(s)\xi_s = \sum_{s\in S}\left(\sum_{t\in S} P_{s,t}R(s,t)\cdot\sum_{k=0}^{\infty} P_{s_0,s}\right)$$

Since for the leakage reward function $R(s,t)$ is a function of $P_{s,t}$, the transition probabilities are the only variables in the equation.

In the example in Fig. 7 the leakage is equal to the entropy, so the reward function is $R(s,t) = -\log_2 P_{s,t}$ and the leakage equation is

$$\begin{aligned}
\mathcal{R}(\mathcal{C}) = &-(\mathbf{P}(0)/4 + \mathbf{P}(\neg 0)/2)\log((\mathbf{P}(0)/4 + \mathbf{P}(\neg 0)/2)) - \\
&-(3\mathbf{P}(0)/4 + \mathbf{P}(\neg 0)/2)\log((3\mathbf{P}(0)/4 + \mathbf{P}(\neg 0)/2))
\end{aligned} \quad (3)$$

under the constraint above.

Step 4: Maximize the leakage equation. Maximizing the extracted constrained leakage equation computes the channel capacity of the system. This can be done with any maximization method. Note that in general the strategy maximizing this reward function will be probabilistic, and thus will have to be approximated

numerically. In the cases in which the maximum leakage strategy is deterministic, an analytical solution can be defined via Bellman equations. This case is more complex that standard reward maximization for MDPs, since the strategy in every state must depend on the same prior information $\mathcal{I}_{\mathcal{A}}$, and this is a global constraint that cannot be defined in a MDP. A theoretical framework to automate this operation is being studied, but most cases are simple enough to not need it, like the examples in the next Section.

9. Onion Routing

9.1. Case: Channel Capacity of Onion Routing

Onion Routing [9] is an anonymity protocol designed to protect the identity of the sender of a message in a public network. Each node of the network is a router and is connected to some of the others, in a directed network connection topology; the topology we consider is the depicted in Fig. 8. When one of the nodes in the topology wants to send a message to the receiver node R, it initializes a path through the network to route the message instead of sending it directly to the destination. The node chooses randomly one of the possible paths from itself to R, respecting the following conditions:

1. No node can appear in the path twice.
2. The sender node cannot send the message directly to the receiver.
3. All paths have the same probability of being chosen.

If some nodes are under the control of an attacker, he may try to gain information about the identity of the sender. In this example node 3 is a compromised node; the attacker can observe the packets transitioning through it, meaning that when a message passes through node 3 the attacker learns the previous and next node in the path. The goal of the attacker is to learn the identity of the sender of the message; since there are 4 possible senders, this is a 2-bit secret. For simplicity, we assume that the attacker knows that a packet is passing through the onion routing system. This is consistent with similar analyses of the Onion Routing protocol [5].

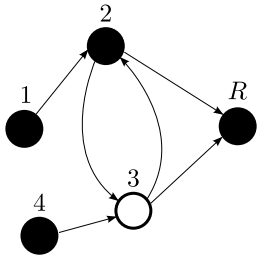

Figure 8: Network topology for Onion Routing

$\mathtt{h}$	Path	$\mathtt{o}$	$\mathrm{P}(\mathtt{O}\|\mathtt{h})$
$1(h_1)$	$1 \to 2 \to R$	NN	$1/2$
	$1 \to 2 \to 3 \to R$	$2R$	$1/2$
$2(h_2)$	$2 \to 3 \to R$	$2R$	1
$3(h_3)$	$3 \to 2 \to R$	$N2$	1
$4(h_4)$	$4 \to 3 \to R$	$4R$	$1/2$
	$4 \to 3 \to 2 \to R$	42	$1/2$

Figure 9: Onion Routing paths, observations and probabilities

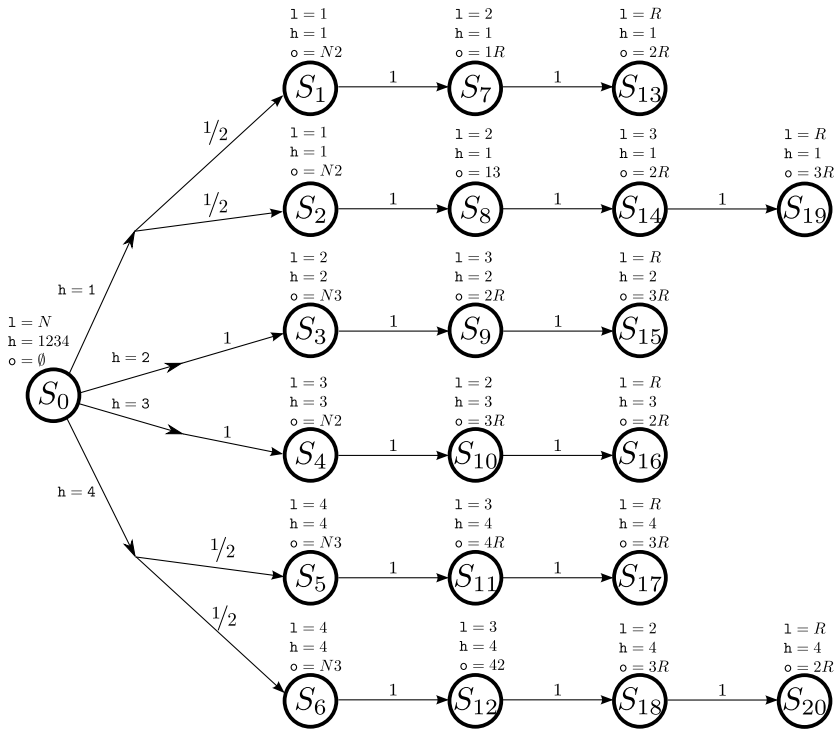

Figure 10: Markov Decision Process for Onion Routing

Figure 9 summarizes the possible secrets of the protocol, the corresponding paths, the observation for each path assuming node 3 is compromised and the probability that a given sender will choose the path.

We give directly the MDP semantics of the system in Fig. 10. The prior information $\mathcal{I}_{\mathcal{A}}$ of the attacker consists of the prior probabilities he assigns to the identity of the sender; we use h_i to denote $\mathbf{P}(\mathtt{h} = i)$, for $i = 1...4$. Clearly $h_1 + h_2 + h_3 + h_4 = 1$. The full system is represented in Fig. 10, parametrized on the h_i parameters. Each state is labeled with the low-level variables $\mathtt{l}$ and $\mathtt{o}$ and the confidential variable $\mathtt{h}$. Variable $\mathtt{l}$ represents the name of the node being visited in the Onion Routing topology, $\mathtt{o}$ represents the observables in that node (the nodes before and after it in the path), and $\mathtt{h}$ the name of the sender of the message.

Since the attacker can observe only node 3, all states with $\mathtt{l} \neq 3$ except the initial state are hidden states. We reduce the chain accordingly; the resulting observational reduction is shown in Fig. 11a. We call it $\mathbb{C}$. Note that one of the paths does not pass through node 3, so if that path is chosen the attacker will never observe anything; in that case the system diverges. We assume that the attacker can recognize this case, using a timeout or similar means.

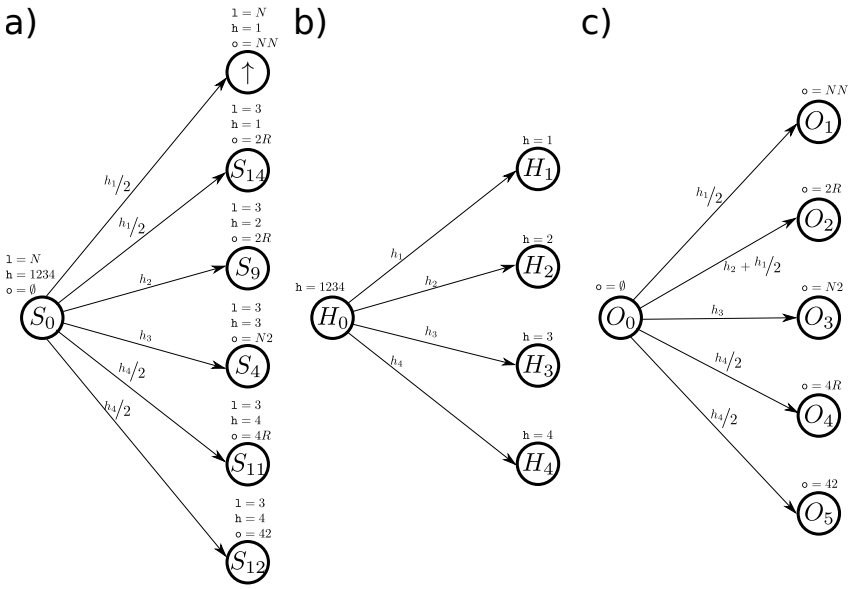

Figure 11: Markov chains for Onion Routing: a) Observable reduction $\mathbb{C}$ b) $\mathbb{C}/\mathcal{R}_h$ c) $\mathbb{C}/\mathcal{R}_{\mathcal{A}}$

To compute the leakage we need also to define $\mathcal{R}_h$ and $\mathcal{R}_{\mathcal{A}}$. This is straightforward; $\mathcal{R}_h$ is $((s,t) \in (S \times S) | \mathtt{h}_s = \mathtt{h}_t)$ and $\mathcal{R}_{\mathcal{A}}$ is $((s,t) \in (S \times S) | \mathtt{o}_s = \mathtt{o}_t)$. The resulting MCs $\mathbb{C}_h = \mathbb{C}/\mathcal{R}_h$ and $\mathbb{C}_O = \mathbb{C}/\mathcal{R}_{\mathcal{A}}$ are shown in Fig. 11bc. Note that $\mathbb{C}_{O,h} = \mathbb{C}/\mathcal{R}_h \cap \mathcal{R}_{\mathcal{A}} = \mathbb{C}$.

Since the system is very simple, we can extract the leakage equation directly from Def. 9. The leakage parametrized on $\mathcal{I}$ is

$$\begin{aligned} & H(\mathbb{C}_h) + H(\mathbb{C}_O) - H(\mathbb{C}_{O,h}) = \\ & = H(h_1, h_2, h_3, h_4) + H(\frac{h_1}{2}, \frac{h_1}{2} + h_2, h_3, \frac{h_4}{2}, \frac{h_4}{2}) - \\ & \quad H(\frac{h_1}{2}, \frac{h_1}{2}, h_2, h_3, \frac{h_4}{2}, \frac{h_4}{2}) \end{aligned} \tag{4}$$

Under constraints $0 \leq h_i \leq 1$ and $h_1 + h_2 + h_3 + h_4 = 1$ it has its maximum of 1.819 bits at $h_1 = 0.2488$, $h_2 = 0.1244$, $h_3 = 0.2834$, $h_4 = 0.2834$, thus these are the channel capacity and the attacker with highest leakage.

9.2. Case: Channel Capacity of Discrete Time Onion Routing

Due to our intensional view of the system, we can naturally extend our analysis to integrate timing leaks. Time-based attacks on the Tor implementation of the Onion Routing network have been proven to be effective, particularly in low-latency networks [10, 11]. We show how to quantify leaks for an attacker capable to make some timing observations about the network traffic.

In this example there are two compromised nodes, A and B, and the attacker is able to count how many time units pass between the message being forwarded

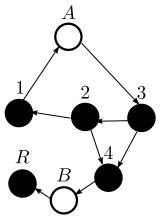

Figure 12: Network topology for Timed Onion Routing

h	Path	o	P(O|h)
$1(h_1)$	$1 \to A \to 3 \to 4 \to B \to R$	$13, 4R$	$1/2$
	$1 \to A \to 3 \to 2 \to 4 \to B \to R$	$13, 4R$	$1/2$
$2(h_2)$	$2 \to 4 \to B \to R$	$NN, 4R$	$1/2$
	$2 \to 1 \to A \to 3 \to 4 \to B \to R$	$13, 4R$	$1/2$
$3(h_3)$	$3 \to 4 \to B \to R$	$NN, 4R$	$1/2$
	$3 \to 2 \to 4 \to B \to R$	$NN, 4R$	$1/2$
$4(h_4)$	$4 \to B \to R$	$NN, 4R$	1

Figure 13: Timed Onion Routing paths, observations and probabilities

by A and the message arriving in B. The topology of the network is shown in Fig. 12 and the relative paths, observations and probabilities in Fig. 13. We will ignore messages departing from the compromised nodes A and B for simplicity.

We add to the system a low-level variable $\mathtt{t}$ that represents the passage of the time between the message passing by A and passing by B. Variable $\mathtt{t}$ is initialized to 0 when the message passes by A and increased by 1 at each subsequent step. We will analyze the difference of leakage between the attacker $\mathcal{A}_\mathcal{T}$ that can discriminate states with different values of $\mathtt{t}$ and the attacker $\mathcal{A}_\mathcal{N}$ that does not have this power. Whenever a packet passes by one of the compromised nodes A and B the attacker observes what node sent the package and to what node the package is to be forwarded next, and the value of the time variable $\mathtt{t}$. Since the attacker can perform two different observations at two different times (when the packet passes through A and when it passes through B), the observable reduction in Fig. 14a is not a one-step Markov chain. In general this could mean that the quotients are not Markov chains. However in this case it is easy to check that they are.

Both attackers are able to observe nodes A and B, so they have the same hidden states. Their observable reduction $\mathbb{C}$ of the system is the same, depicted in Fig. 14a. The secret's discrimination relation is also the same: $\mathcal{R}_h$ is $((s,t) \in (S \times S) | \mathtt{h}_s = \mathtt{h}_t)$, and the resulting quotient $\mathbb{C}_h = \mathbb{C}/\mathcal{R}_h$ is depicted in Fig. 14b.

The two attackers have two different discrimination relations. For the attacker $\mathcal{A}_\mathcal{N}$, who is not able to keep count of the discrete passage of time, the relation is $\mathcal{R}_{\mathcal{A}_\mathcal{N}} = ((s,t) \in (S \times S) | \mathtt{o}_s = \mathtt{o}_t)$, while for the time-aware attacker $\mathcal{A}_\mathcal{T}$ it is $\mathcal{R}_{\mathcal{A}_\mathcal{T}} = ((s,t) \in (S \times S) | \mathtt{o}_s = \mathtt{o}_t \wedge \mathtt{t}_s = \mathtt{t}_t)$. The resulting MCs $\mathbb{C}_{O_\mathcal{N}} = \mathbb{C}/\mathcal{R}_{\mathcal{A}_\mathcal{N}}$ and $\mathbb{C}_{O_\mathcal{T}} = \mathbb{C}/\mathcal{R}_{\mathcal{A}_\mathcal{T}}$ are shown in Fig. 15.

Note that since the time-aware attacker has strictly more discriminating power, since $\mathcal{R}_{\mathcal{A}_\mathcal{T}} \subseteq \mathcal{R}_{\mathcal{A}_\mathcal{N}}$, we expect that he will leak more information. We show now how to validate this intuition by computing the difference of the leakage between $\mathcal{A}_\mathcal{T}$ and $\mathcal{A}_\mathcal{N}$. The difference of the leakage between the two

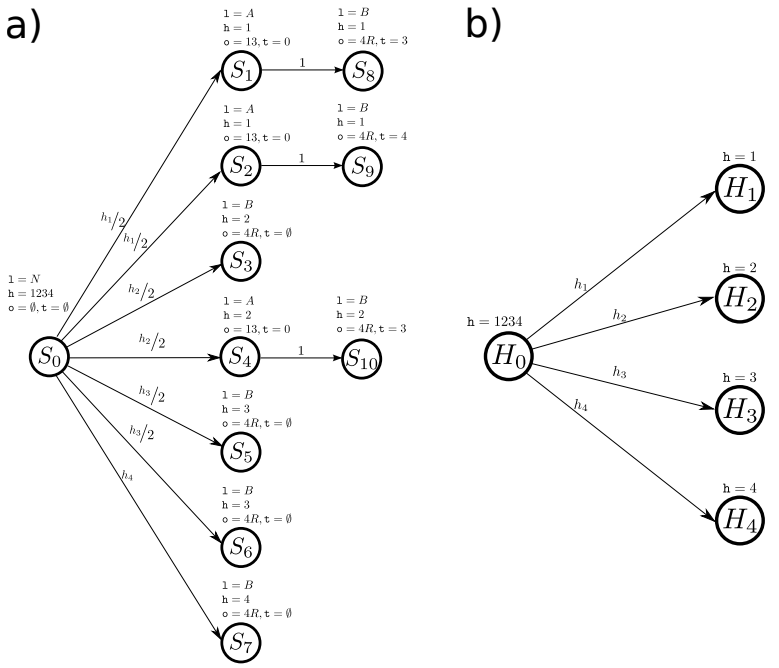

Figure 14: Markov chains for Timed Onion Routing: a) Observable reduction $\mathbb{C}$ b) $\mathbb{C}_h$

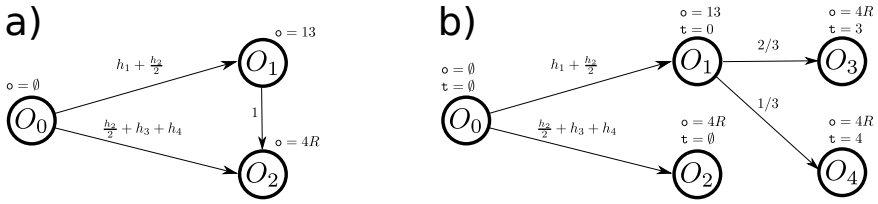

Figure 15: Markov chains for Timed Onion Routing: a) $\mathbb{C}_{O_\mathcal{N}}$ b) $\mathbb{C}_{O_\mathcal{T}}$

attackers is

$$
\begin{aligned}
&I(\mathbb{C}_h;\mathbb{C}_{O_\mathcal{T}}) - I(\mathbb{C}_h;\mathbb{C}_{O_\mathcal{N}}) = \\
&H(\mathbb{C}_h) + H(\mathbb{C}_{O_\mathcal{T}}) - H(\mathbb{C}_{(O,h)_\mathcal{T}}) - H(\mathbb{C}_h) - \\
&\qquad - H(\mathbb{C}_{O_\mathcal{N}}) + H(\mathbb{C}_{(O,h)_\mathcal{N}}) = \\
&H(\mathbb{C}_{O_\mathcal{T}}) - H(\mathbb{C}_{O_\mathcal{N}}) = \\
&H\left(h_1 + \frac{h_2}{2}, \frac{h_2}{2} + h_3 + h_4\right) + \left(h_1 + \frac{h_2}{2}\right) H\left(\frac{1}{3}, \frac{2}{3}\right) - \\
&\qquad - H\left(h_1 + \frac{h_2}{2}, \frac{h_2}{2} + h_3 + h_4\right) = \\
&\left(h_1 + \frac{h_2}{2}\right) H\left(\frac{1}{3}, \frac{2}{3}\right) \approx \\
&0.91829\left(h_1 + \frac{h_2}{2}\right)
\end{aligned}
\tag{5}
$$

showing that the time-aware attacker $\mathcal{A}_{\mathcal{T}}$ leaks $\approx 0.91829\left(h_1 + \frac{h_2}{2}\right)$ bits of information more than the time-unaware attacker $\mathcal{A}_{\mathcal{N}}$.

10. The QUAIL Preliminary Implementation

As a proof of concept we present an implementation of part of the technique presented in Section 3, which we called QUantitative Analyzer for Imperative Languages (QUAIL). In particular, QUAIL is able to compute the information leakage of a randomized program written in an imperative language when observed by an ignorant attacker, i.e. an attacker whose prior distribution $\mathcal{I}_{\mathcal{A}}$ over the possible values of the secret is uniform. It models the case in which the attacker can observe the values of the observable variables after the termination of the program. If the program does not necessarily terminate, it is assumed that the non-termination of the program is one of the possible observable outcomes.

QUAIL does not allow the user to define an arbitrary prior distribution for the attacker, because of the inherent complexity of representing arbitrary probability distribution over large sample spaces of exponential size in the size of the variables. For similar reasons, QUAIL does not implement the computation of channel capacity presented in Section 8.

QUAIL is available at `https://project.inria.fr/quail/`.

10.1. QUAIL Imperative Language

QUAIL supports a simple but powerful imperative WHILE language. The language includes constants and array declarations, but not function declarations.

10.1.1. Variable declarations

All variables in QUAIL are fixed sized integers. We force them to be declared at the beginning of the program to ensure that the analysis terminates and to simplify scoping.

Variables are declared in one of the four following types: *public, private, secret* and *observable*.

Public and observable variables have an explicit value during the computation. They represent variables whose value is known at computation time. The only difference between public and observable variables is that the attacker can discriminate states that have different values of the latter but not of the former. Public and observable variables can be manipulated using standard arithmetic operators.

Public and observable variables are declared as follows:

```
public int4 v;
```

or

```
observable int4 v;
```

declares a 4 bits integer variable whose name is `v`, either public or observable.

```
public int4 v := 5;
```

declares `v` and initializes it to value 5. Any expression can be used to initialize a variable, provided that the variables used in the expression are public or constants and have been previously declared. Since QUAIL does not support function declaration there is a single variable scope. Declaring multiple variables with the same name is consequently forbidden.

Private and secret variables represent variables that do not have a known fixed value, but instead a uniform probability distribution over a set of values. The set of values is represented as a sequence of integer intervals, thus for instance the set $\{0, 1, 2, 3, 5, 6, 8, 9, 10, 13\}$ would be represented as $[0, 3][5, 6][8, 10][13, 13]$.

The difference between private and secret variables is that the information leakage is only computed on the latter, while knowing the exact value of the former is considered to be information that is not interesting to the attacker.

Private and secret variables cannot be used in assignment and expressions. They can only appear to the left of the operator in a guard. This is to restrict the leakage analysis to indirect flow of information, but direct flow can still be modeled if necessary.

Private and secret variables are declared as follows:

```
private int4 v;
```

or

```
secret int4 v;
```

declares a 4 bits integer variable whose name is `v`, either private or secret.

```
private int4 v := [0,1][2,5];
```

declares `var` and restricts its range to the two intervals [0,1] and [2,5]. Again any expression can be used in the bounds of the intervals.

QUAIL allows for the declaration of integer constants. Constants are declared as follows:

```
const N := 4;
```

They are replaced by their value during the preprocessing step.

10.1.2. Arrays

Variables can also be arrays of integers and multi-dimensional arrays. Arrays are declared before the integer type of a variable.

```
public array[7] of int4 tab;
```

declares a public variable `tab` that is an array of 4 bits integer of size 7 whose indexes range from 0 to 6, while

```
public array[1..7] of int4 tab;
```

declares `tab` as an array of size 4 whose indexes range from 1 to 7. The size of an array can be any expression that evaluates to an integer.

An array may be initialized with a set of initial values:

```
public array[1..4] of int4 tab := {1,1,2,2};
```

initializes `tab` such that `tab[1]` and `tab[2]` are equal to 1, while `tab[3]` and `tab[4]` are equal to 2. Private arrays can be initialized like any private variable, with a set of intervals:

```
private array[1..4] of int4 tab := [0,1];
```

In that case all the variables in the array are initialized to the same range of integers.

10.1.3. Expressions

Expressions are used in guards, assignments, variables initialization and arrays indexes. Binary operators (`||`,`&&`,`^`,`+`,`-`,`*`,`/` and `%`) and unary operators (`-`,`!`) can used. Classical operators precedence is assumed. For Boolean operations integer variables are considered as a true value if non null, and false if null. Only public and observable variables, constants and integers can be used in expressions.

10.1.4. Guards

Guards are limited to a single comparison between a variable on the left side (either public, or private, or constant, or an integer value) and an expression on the right side. Any comparison operator among `<`,`>`,`<=`,`>=`,`==` and `!=` can be used.

10.1.5. Assignments

An assignment statement is written in the following manner:

```
assign v := expr;
```

where `v` is a public or observable variable (possibly with indexes) and *`expr`* is an expression containing no private or secret variables.

10.1.6. Random assignments

The program can use two types of random primitives to assign values to a variable.

```
random v := random(expr_min,expr_max);
```

assigns to a public variable `v` a random value, chosen between the values of *`expr_min`* and *`expr_max`*, with a uniform probability distribution.

```
random v := randombit(p);
```

where `p` is a float value lower than 1, assigns to a public variable `v` a random bit value, that is 0 with probability `p`, and 1 with probability $1 - \texttt{p}$.

10.1.7. IF statements

IF conditional statements starts with the keyword `if`, possibly followed by `elif` and `else`, and ends with `fi`. The consequent statements are listed after the keyword `then`. For example the following structures are allowed:

```
if (h <= l) then assign v:=1;
fi

if (h <= l) then assign v:=1;
else assign v:=2;
fi

if (h <= l) then assign v:=1;
elif (h == l) then assign v:=2;
fi

if (h <= l) then assign v:=1;
elif (h == l) then assign v:=2;
elif (h == l+1) then assign v:=3;
else assign v:=4;
fi
```

10.1.8. WHILE statements

Conditional WHILE loop starts with the keyword `while`, followed by a guard, and the statements included in the loop are listed between the keywords `do` and `od`. For example the following structure is allowed:

```
while (h <= l) do
assign l := 1;
assign v := 2;
od
```

10.1.9. FOR statements

A FOR loop can be used to iterate over all the elements of an array. The syntax is:

```
for (v in tab) do
assign v := v+1;
od
```

The variable `v` is a local variable that must only be used inside the loop. It will take successively each value in the array tab. Note that if `tab` is a multi-dimensional array `v` is also an array.

10.1.10. Return statements

The program ends when a return statement is reached. Its syntax is simply:

```
return;
```

10.2. Attacker Encoding

QUAIL assumes that the attacker is ignorant, i.e. does not have any information about the value of the secret except in which range it is, e.g. from 0 to 7 for a 3-bit secret. Like all information-theoretical analysis we assume that the attacker has access to the source code of the system: assuming otherwise would invalidate the analysis in case the attacker was able to obtain or infer information about such code.

The assumption allows QUAIL to build directly the Markov chain model of the scenario, since whenever a prior distribution on the secret is encountered it can be assumed to be uniform.

The attacker is assumed to be able to start the program and observe the values of the output variables after the program's termination. For this reason, all and only the internal states of the Markov chain are hidden during the hiding part of the modeling (see Section 5). If the program does not terminate, we assume that the attacker is able to recognize this, e.g. via a timeout. In this case the attacker knows that the program did not terminate but is not able to read any variable's value.

For the discrimination relation, as we said the attacker is assumed to be able to observe only a given subset of the variables, that we call *observable* variables. The attacker can thus discriminate two states if and only if they differ in the value of any of the observable variables, while different states that assign the same values to the observable variables are impossible to discriminate for him.

To encode an attacker, the QUAIL user only has to specify which variables are the secret and which variables are observable to the attacker. The rest of the encoding is automatically handled by the tool.

10.3. Procedure

QUAIL's analysis proceeds as explained in Section 3, with some improvements to make the process more streamlined and implementable.

Step 1: Preprocessing. In this step the imperative code gets rewritten in a simplified `if-goto` language, following common compiler practice. All `if-elif-else-fi` conditional statements and `while` loop statements are rewritten as follows:

- if-elif-else-fi statement

```
1 if CONDITION1 then STATEMENT1;
2 elif CONDITION2 then STATEMENT2;
3 else STATEMENT3;
4 fi
```

  becomes

```
1 if CONDITION1
2 then goto 4;
3 else goto 6;
4 STATEMENT1;
```

```
5  goto 12;
6  if CONDITION2
7  then goto 9;
8  else goto 11;
9  STATEMENT2;
10 goto 12;
11 STATEMENT3;
12 ...
```

- `while` statement

```
1 while CONDITION do
2 STATEMENT
3 od
```

becomes

```
1 if CONDITION
2 then goto 4
3 else goto 6
4 STATEMENT
5 goto 1
6 ...
```

Also, array calls are substituted with single indexed variables and `for` statements rewritten to a sequence of commands on such variables. Finally, constants are substituted with their value. In this step we also add automatically a `free` command to signal when a variable is not used anymore and can be collected.

Step 2: Probabilistic Symbolic Execution. QUAIL symbolically executes the preprocessed code and builds an annotated Markov chain semantics of the program execution, as explained in Section 3. This step includes building the MDP model of the system and applying the prior information of the attacker to it: since we assume that the prior probability distribution over the values of the secret is uniform, encoding the ignorant attacker, we can build the Markov chain directly. Whenever a conditional guard is found QUAIL has sufficient information to compute the probability that the guard will be satisfied, and constructs two successor states, one if the guard is true and one if it is false, with appropriate transition probabilities. This is the most time-consuming step of the computation, since it requires building a full control flow graph of the system's behavior and assigning probabilities to it. In the worst case the graph has exponential size in the size of the variables, but for most academic cases it has a reasonable size of thousands or tens of thousands of nodes. We apply on-the-fly reduction techniques like avoiding producing states that we know will be removed by the next step, but these do not significantly reduce the computation time of this step.

Step 3: State Hiding and Model Reduction. QUAIL assumes that the attacker can only observe the values of some variables at the end of the computation, thus all internal states of the system are to be hidden. We apply Algorithm 1

iteratively on all internal states until only the initial state and the output states remain. In this step we also detect nonterminating behavior and, if needed, we construct an output state modeling non-termination. This operation removes more than 90% of the states of the Markov chain model, in fact producing a Markov chain with a single probability distribution from the initial state to the output states. To make this operation as quick as possible, states are equipped with a list of their predecessors and successors.

Step 4: Quotienting. In this step we construct three quotients of the Markov chain, as explained in Section 3. A quotient is obtained by merging together the states that correspond to the same equivalence class in a given equivalence relation, as explained in Section 2. The quotients are as follows:

- The *attacker's quotient* represents the view that the attacker has of the system. It is obtained by merging together states that assign the same values to all and only the observable variables.
- The *secret's quotient* represents the system as it depends on the secret. Its entropy is a measure of how much of the secret is actually used in the execution of the program. It is obtained by merging together states that assign the same sets of values to all and only the secret variables.
- The *joint quotient* represents the joint behavior of the observable variables and secret variables. It is obtained by merging together states that assign the same values and sets of values to the observable and secret variables.

To speed up the process, QUAIL drops the information about the variable assignments in the states of the quotients. Such information is not needed to perform the rest of the analysis.

Step 5: Entropy and Leakage Computation. Finally, QUAIL computes the entropy of the three quotients in linear time in the size of the quotients. The three computations are independent and can be parallelized. The information leakage is then computed as the sum of the entropies of the attacker's and secret's quotients minus the entropy of the joint quotient, as explained in Section 3. QUAIL outputs the result with the requested amount of significant digits. If requested, the tool also prints any of the Markovian models it has produced during the analysis.

10.4. Case Studies

We show how to analyze a number of protocols and academic examples with QUAIL. For each example we show the commented source code we used to encode it and comment on the results and variants.

```
1  // this bit is observable
       by the user; it is 0
       for REJECT and 1 for
       ACCEPT
2  observable int1 o;
3
4  // this represents the
       password inserted by
       the user
5  public int2 input:=2;
6
7  // this is the secret
8  secret int2 password;
9
10 //
11 if (password==input) then
12   assign o:=1;
13 else
14   assign o:=0;
15 fi
16
17 // terminate
18 return;
```

Password length	Leakage
1	1
2	$8.11 \cdot 10^{-1}$
32	$7.78 \cdot 10^{-9}$
64	$3.54 \cdot 10^{-18}$
512	$3.81 \cdot 10^{-152}$

Figure 16: Simple authentication example: model (on the left) and resulting leakage according to password length (on the right).

10.4.1. Simple Authentication

In this basic example the attacker is trying to infer the password of a system by trying to provide as a password a given number in the password domain. The model is shown in Fig. 16 on the left.

The length of the secret is the size of the variable `password` on line 8. Since the variable is not explicitly initialized, QUAIL assumes that it is in the interval $[0, 2^{size} - 1]$. On line 5 the variable `input`, representing what is input by the attacker, is initialized with an arbitrary value. The value chosen is irrelevant as long as it is in the range of the possible values for the password.

Increasing the size of the password changes the amount of leakage, since the attacker learns less information about the password from a single attack attempt. The information leakage values for some password sizes are shown in Fig. 16 on the right. Since this quantifies Shannon leakage, the leakage value is inversely proportional to the time it takes to learn the secret by brute force. The results prove that it is harder for an attacker to brute force a password with a larger password space, in accordance with intuition.

It is worth noting that QUAIL solves this example with any password size in milliseconds because it uses the Markovian process encoding that we presented in this work. In particular, the size of the Markov chain in this case does not depend on the size of the secret. Any analysis based on channel matrices would have to build the channel matrix for this examples. Such matrix has a number of rows exponential in the size of the secret. This operation alone would require from days to millennia, according to the chosen size of the password. While the Markovian model we presented in general has an exponential number of states in the size of the secret and of the observable, this example shows how in many

```
1  observable int1 o; // this
       bit represents the
       output
2
3  public int1 r; // this bit
       is randomly generated
4
5  // this bit is the secret
6  secret int1 s;
7
8  // randomize the random bit
       with a given
       probability
9  random r:=randombit(0.5);
10
11 // calculate the XOR
12 if (s==r) then
13 assign o:=0;
14 else
15 assign o:=1;
16 fi
17
18 // terminate
19 return;
```

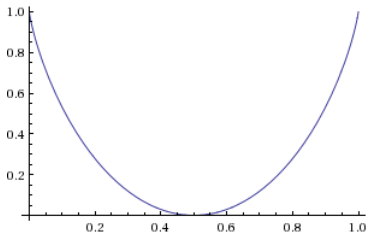

Figure 17: Bit XOR example: model (on the left) and graph of the information leakage over the probability of the random bit on line 9 (on the right).

useful cases the model is significantly smaller than the worst case. Again this is an improvement compared to channel matrices, whose size is always exponential in the size of the secret and observables, and not only in the worst case.

We finally remark that the results are presented with 2 decimal digits, but QUAIL can work with any precision, as requested by the user.

10.4.2. Bit XOR

This is one of the simplest examples of randomized programs depending on a secret. In this case the secret is a bit. The system produces a random bit with a probability distribution known to the attacker, computes the exclusive OR of the secret and the random bit, and outputs the result to the attacker. The question is how much of the secret bit can the attacker infer by knowing the result of the exclusive OR. The model is shown in Fig. 17 on the left.

On line 9 we assign to the random bit `r` the value 0 with the given probability and 1 otherwise. We remind that the attacker possesses the source code, so he knows the probability distribution over `r`, but not the value that gets assigned to `r` during a given execution.

Note that we cannot directly calculate the result of the exclusive or by writing `assign o := s XOR r` because QUAIL does not allow private or secret variables to appear in an assignment statement. This is a limit of the representation of the variables in the tool, not in the theory. Work is under way to allow for a more natural encoding of this kind of operations.

```
1  // a variable to
       loop on
2  observable int2
       o:=0;
3
4  // this is the
       secret,
       either 0, 1,
       2 or 3
5  secret int2 s
       :=[0,3];
6
7  // if the secret
       is 0 then
       loop forever
8  if (s==0) then
9    while (o==0)
         do
10   skip;
11   od
12 fi
13
14 // if the secret
       is 3 also
       loop forever
15 if (s==3) then
16   while (o==0)
         do
17   skip;
18   od
19 fi
20
21 // terminate
22 return;
```

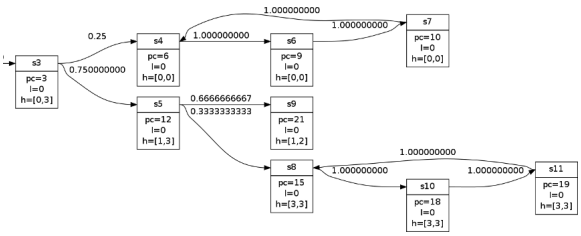

Figure 18: Conditional non-termination example: model (on the left) and snippet of the Markov chain model (on the right).

10.4.3. Conditional Non-Termination

This example shown QUAIL's treatment of non-terminating programs. We have a 2-bit secret variable `s`, i.e. `s` is either 0, 1, 2 or 3. There is also an observable variable `o` that is initialized to 0 and never changes its value. Then the program terminates if `s` is 1 or 2, and loops forever otherwise. The model for the example is shown in Fig. 18 on the left.

In this example QUAIL reports a leakage of 1 bit. The reason is that the attacker, being able to distinguish whether the program terminates or not via a timeout, can infer whether the secret is 1 or 2 or whether is 0 or 3. In both cases its ignorance goes down from 4 possible cases to 2 possible cases, thus being quantified in 1 bit of gained information.

In Fig. 18 on the right we can see the last part of the Markov chain produced by QUAIL, in particular showing how QUAIL detects non-terminating loops.

11. Related work

Alvim, Andrés and Palamidessi [28] study leakage and channel capacity of interactive systems where secrets and observables can alternate during the computation.

Chen and Malacaria study leakage and channel capacity of traces and subtraces of programs [29], and, in [30], consider transition systems with particular attention to multi-threaded programs. They use Bellman equations to determine the minimal and maximal leakage. None of these works however deal explicitly with Markov Chains and randomized systems.

Intensional aspects of systems like timing leaks have been investigated by Köpf et al. in [7, 6, 31] and more recent work by Köpf, Mauborgne and Ochoa has investigated caching leaks [32].

Channel capacity for the Onion Routing protocol has been first characterized by Chen and Malacaria using Lagrange multipliers [5].

The Lattice of Information approach to security seems to be related to the Abstract Interpretation approach to code obfuscation investigated by Giacobazzi et al. [33]; it would be interesting to further understand the connection between these approaches.

12. Conclusion

We presented a method to quantify the information leakage of a probabilistic system to an attacker. The method considers the probabilistic partial information semantics of the system and allows to encode attackers that can partially observe the internal behavior of the system. The method presented can be fully automated, and an implementation is being developed. The paper extends the consolidated LoI approach for leakage computation to programs with randomized behavior.

We extended the method to compute the channel capacity of a program, thus giving a security guarantee that does not depend on a given attacker, but considers the worst case scenario. We show how this can be obtained by maximizing an equation parametrized on the prior information of the attacker. The automatization of this computation raises interesting theoretical problems, as it requires to encode the property that all probability distributions on state must be derived from the same prior information, and thus involves a global constraint. We intend to work further on identifying suitable optimizations for constraints arising in this problem.

Finally, we analyzed the channel capacity of the Onion Routing protocol, encoding the classical attacker able to observe the traffic in a node and also a new attacker with time-tracking capabilities, and we proved that the time-tracking attacker is able to infer more information about the secret of the system.

References

[1] Malacaria, P.: Algebraic foundations for information theoretical, probabilistic and guessability measures of information flow. CoRR **abs/1101.3453** (2011)

[2] Clark, D., Hunt, S., Malacaria, P.: A static analysis for quantifying information flow in a simple imperative language. Journal of Computer Security **15** (2007) 321–371

[3] Heusser, J., Malacaria, P.: Quantifying information leaks in software. In Gates, C., Franz, M., McDermott, J.P., eds.: ACSAC, ACM (2010) 261–269

[4] Chatzikokolakis, K., Palamidessi, C., Panangaden, P.: Anonymity protocols as noisy channels. Inf. Comput. **206** (2008) 378–401

[5] Chen, H., Malacaria, P.: Quantifying maximal loss of anonymity in protocols. In Li, W., Susilo, W., Tupakula, U.K., Safavi-Naini, R., Varadharajan, V., eds.: ASIACCS, ACM (2009) 206–217

[6] Köpf, B., Smith, G.: Vulnerability bounds and leakage resilience of blinded cryptography under timing attacks. [34] 44–56

[7] Köpf, B., Basin, D.A.: An information-theoretic model for adaptive side-channel attacks. In Ning, P., di Vimercati, S.D.C., Syverson, P.F., eds.: ACM Conference on Computer and Communications Security, ACM (2007) 286–296

[8] Millen, J.K.: Covert channel capacity. In: IEEE Symposium on Security and Privacy. (1987) 60–66

[9] Goldschlag, D.M., Reed, M.G., Syverson, P.F.: Onion routing. Commun. ACM **42** (1999) 39–41

[10] Murdoch, S.J., Danezis, G.: Low-cost traffic analysis of tor. In: Proceedings of the 2005 IEEE Symposium on Security and Privacy. SP '05, Washington, DC, USA, IEEE Computer Society (2005) 183–195

[11] Abbott, T.G., Lai, K.J., Lieberman, M.R., Price, E.C.: Browser-based attacks on tor. In Borisov, N., Golle, P., eds.: Privacy Enhancing Technologies. Volume 4776 of Lecture Notes in Computer Science., Springer (2007) 184–199

[12] Applebaum, D.: Probability and Information: An Integrated Approach. Cambridge University Press, New York, NY, USA (2008)

[13] Cover, T.M., Thomas, J.A.: Elements of information theory (2. ed.). Wiley (2006)

[14] Shannon, C.E.: A mathematical theory of communication. The Bell system technical journal **27** (1948) 379–423

[15] Biondi, F., Legay, A., Nielsen, B.F., Wasowski, A.: Maximizing entropy over markov processes. In Dediu, A.H., Martín-Vide, C., Truthe, B., eds.: LATA. Volume 7810 of Lecture Notes in Computer Science., Springer (2013) 128–140

[16] Smith, G.: Quantifying information flow using min-entropy. In: QEST, IEEE Computer Society (2011) 159–167

[17] Smith, G.: On the foundations of quantitative information flow. In de Alfaro, L., ed.: FOSSACS. Volume 5504 of Lecture Notes in Computer Science., Springer (2009) 288–302

[18] Backes, M., Köpf, B., Rybalchenko, A.: Automatic discovery and quantification of information leaks. In: IEEE Symposium on Security and Privacy, IEEE Computer Society (2009) 141–153

[19] Alvim, M.S., Chatzikokolakis, K., Palamidessi, C., Smith, G.: Measuring information leakage using generalized gain functions. In Chong, S., ed.: CSF, IEEE (2012) 265–279

[20] Boreale, M., Pampaloni, F.: Quantitative information flow under generic leakage functions and adaptive adversaries. In Ábrahám, E., Palamidessi, C., eds.: FORTE. Volume 8461 of Lecture Notes in Computer Science., Springer (2014) 166–181

[21] Landauer, J., Redmond, T.: A lattice of information. In: CSFW. (1993) 65–70

[22] Yasuoka, H., Terauchi, T.: Quantitative information flow - verification hardness and possibilities. [34] 15–27

[23] Winskel, G.: The formal semantics of programming languages - an introduction. Foundation of computing series. MIT Press (1993)

[24] Malacaria, P.: Risk assessment of security threats for looping constructs. Journal of Computer Security **18** (2010) 191–228

[25] McIver, A., Meinicke, L., Morgan, C.: Compositional closure for bayes risk in probabilistic noninterference. In Abramsky, S., Gavoille, C., Kirchner, C., Meyer auf der Heide, F., Spirakis, P.G., eds.: ICALP (2). Volume 6199 of Lecture Notes in Computer Science., Springer (2010) 223–235

[26] McIver, A., Morgan, C., Smith, G., Espinoza, B., Meinicke, L.: Abstract channels and their robust information-leakage ordering. In Abadi, M., Kremer, S., eds.: POST. Volume 8414 of Lecture Notes in Computer Science., Springer (2014) 83–102

[27] Nakamura, Y.: Entropy and semivaluations on semilattices. Kodai Mathematical Seminar Reports **22** (1970) 443–468

[28] Alvim, M.S., Andrés, M.E., Palamidessi, C.: Quantitative information flow in interactive systems. Journal of Computer Security **20** (2012) 3–50

[29] Malacaria, P., Chen, H.: Lagrange multipliers and maximum information leakage in different observational models. In ÃŽlfar Erlingsson, Pistoia, M., eds.: PLAS, ACM (2008) 135–146

[30] Chen, H., Malacaria, P.: The optimum leakage principle for analyzing multi-threaded programs. In Kurosawa, K., ed.: ICITS. Volume 5973 of Lecture Notes in Computer Science., Springer (2009) 177–193

[31] Chothia, T., Guha, A.: A statistical test for information leaks using continuous mutual information. In: CSF, IEEE Computer Society (2011) 177–190

[32] Köpf, B., Mauborgne, L., Ochoa, M.: Automatic quantification of cache side-channels. In Madhusudan, P., Seshia, S.A., eds.: CAV. Volume 7358 of Lecture Notes in Computer Science., Springer (2012) 564–580

[33] Preda, M.D., Giacobazzi, R.: Semantics-based code obfuscation by abstract interpretation. Journal of Computer Security **17** (2009) 855–908

[34] Proceedings of the 23rd IEEE Computer Security Foundations Symposium, CSF 2010, Edinburgh, United Kingdom, July 17-19, 2010. In: CSF, IEEE Computer Society (2010)